Mobile Application and Server Management System for Autonomous Robot Deliveries

ASHA KAILIN JAIN

Bachelors of Science in Aerospace Engineering

Supervisor: Dr. Luis Sentis, Associate Professor, Aerospace Engineering

Associate Supervisor: Dr. Junfeng Jiao, Associate Professor, School of Architecture

A thesis submitted in fulfilment of the requirements for

Bachelors of Engineering Honors Thesis

Department of Aerospace Engineering and Engineering Mechanics

Cockrell School of Engineering

The University of Texas at Austin

12 December 2020

Abstract

The Short to Medium Range Autonomous Delivery System (SMADS) is an end-to-end platform to connect UT Austin-affiliated customers to autonomous delivery robots on UT Austin campus. The SMADS system integrates several subsystems, including two iOS mobile applications, a two-server communication structure and a robot autonomy stack with localization and navigation algorithms, into a global system architecture. This thesis describes the steps taken to develop and integrate the customer-facing iOS App, Texas Bolter, the Manager App and the two-server system comprised of the App Server and the robot servers. Furthermore, this work presents the results of the week long deployment of the SMADS system, delivering free lemonades to specific UT Austin buildings. The recorded issues are discussed and future work is presented. The SMADS system successfully completed 27 trips on varying robot platforms to various locations, demonstrating the robot-agnostic nature of the app and server management system — a useful feature for future human-robot interaction research.

Acknowledgements

The Short to Medium Range Autonomous Delivery System project was a highly collaborative research effort with several contributing UT Austin labs, including the Autonomous Mobile Robot Laboratory (AMRL)[1], Human Centered Robotics Lab (HCRL)[2] and the Learning Agents Research Group (LARG)[3]. I want to thank each of the professors and students from these labs who helped shape the development of SMADS.

Thank you Dr. Sentis and Dr. Jiao for supporting me on this project. Your mentorship and advice has been an invaluable contribution to my work.

I also want to thank the the SMADS integration team who put in countless hours to help me integrate the robots with the apps and server. It has been wonderful working with you all. I am grateful to have been your teammate and have contributed to bring this idea to fruition.

Special thanks to Max, Nick, and Cem for your tireless effort to integrate and operate these robots. I enjoyed working alongside you all to solve bugs, test our system and ultimately run a successful lemonade stand. Cheers and best of luck in the future.

[1]https://amrl.cs.utexas.edu
[2]https://sites.utexas.edu/hcrl/
[3]https://www.cs.utexas.edu/ larg/index.php/Learning_Agents_Research_Group

Contents

List of Figures

CHAPTER 1

Introduction

The rise of e-commerce, including services like Amazon[1], has driven a demand for better short to medium range delivery systems which currently include professional mailing services like UPS[2], and crowd-sourced delivery, such as UberEats[3] [1]. Autonomous robots have the potential to improve current transportation systems, delivering goods with less carbon production than current solutions [2]. Furthermore, the COVID-19 pandemic has created an urgent need for efficient contactless delivery systems to reduce rates of virus transmission in public spaces such as restaurants and grocery stores. Autonomous delivery robots recently have been identified as a possible resource in combating COVID-19 by delivering food and medicine to persons at home [3].

Several companies are working to build these autonomous systems to fulfill the aforementioned social need for delivered goods. Fedex recently unveiled its SameDayBot, an autonomous delivery robot intended to make short range deliveries more efficient [4]. Start-ups like Starship Technologies[4] are working to deploy robot systems to deliver food, particularly on college campuses. In Starship's case, the COVID-19 pandemic has skyrocketed the demand, and robots now deliver food, medicine and other essential goods to homes [5].

Motivated by the potential of autonomous robot deliveries, the Short to Medium Range Autonomous Delivery System (SMADS) was developed to create and test an autonomous robot delivery system that navigates the outdoor environment of UT Austin campus. The

[1] https://www.amazon.com
[2] https://www.ups.com/us/en/Home.page
[3] https://www.ubereats.com
[4] https://www.starship.xyz

SMADS team integrated several subsytems, including a customer-facing iOS app, a two-part server structure, localization software and navigation planners, into a global architecture.

To achieve autonomous deliveries, the robot must understand its current location and surrounding environment. These tasks fall under the umbrella of localization, mapping sensed environmental features to a known map. Next, the robot needs to plan a path to its destination and adjust course for obstacles. Navigation assumes these undertakings and reasons about the robots current position relative to the target. While the robot is en-route, the customer should have access to delivery status updates. The Texas Bolter iOS App is responsible for communicating delivery updates to the customer and forwarding customer requests to the system. Finally, these three independent services must work together to ensure that an order reaches a robot and that robot arrives at the intended destination. A sophisticated software architecture coordinates the flow of information, streamlining the communication between the customer and the robot through a two-part server structure.

This thesis specifically explores the app and server development implemented in SMADS to achieve autonomous robot deliveries to customers. The iOS customer-facing app, Texas Botler, provides customers with a single interface in which to interact with SMADS. Currently, customers can order a free lemonade, choose a delivery destination from a selection of UT campus buildings, and watch the delivery robot progress to the drop-off location.

In addition to exploring the development of SMADS mobile applications and two-server system, later chapters specifically address how to information is communicated to and from the robot and customer. This flow is essential for communicating order status updates to the customer and to handle new customer requests. Later chapters discuss the methods employed to secure this human-robot communication and stored data. Furthermore, the thesis presents a detailed discussion on how the two-server system manages the SMADS state, maintaining state consistency across a network of robot systems. Finally, field results from a week-long deployment are presented with future work noted. Future researchers can refer to chapter 9 to review steps on how to step up and run the app and server systems on their local machine.

CHAPTER 2

SMADS System Overview

The Short to Medium Range Autonomous Delivery System (SMADS) was developed to operate an autonomous robot delivery system on UT Austin campus. As a proof of concept, the SMADS team designed the system to deliver free lemonades to a select number of UT campus buildings, namely the Main Tower (MAI) and the Gates Dell Complex (GDC). Customers can place a lemonade order in an iOS app and watch their delivery robot progress towards the drop-off location. This overarching concept drove the design of significant features in the iOS Customer app and the App Server, including the imagery and the order management.

To accomplish said task, SMADS encompasses several subsystems, namely an iOS app for customers and robot managers, several servers, including an app server and robot-specific servers, and a software stack containing code for robot autonomy. More specifically, this autonomy stack includes a robot system architecture which coordinates communication between the localization and navigation software to produce motor commands that ultimately move the robot to a target destination. To orient the reader, a high level overview of these various subsystems and how they interact is shown in Figure 2.1. A brief discussion of each component follows before further chapters detail the app and server systems used to operate the SMADS autonomous delivery system.

2.1 Mobile Applications

To interact with customers, the SMADS framework incorporates a mobile application, called Texas Botler, which is available on the Apple App Store for iOS devices. Referred to as

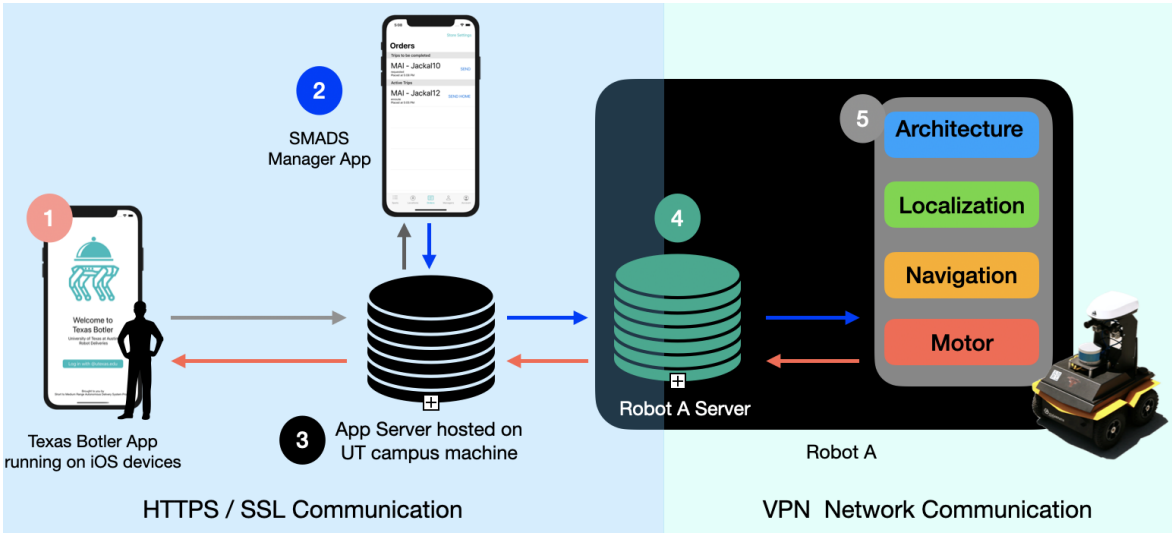


FIGURE 2.1. SMADS System Overview

marker 1 in Figure 2.1, the Texas Botler App gives customers an interface to interact with the SMADS system. Customers can order lemonades, view the current location of their delivery robot, peruse their order history and report complaints regarding a specific delivery. Further details about the Texas Botler App are presented in chapter 4.

In addition to a customer-facing app, SMADS utilizes a separate mobile application for robot managers (labeled with marker 2 in Figure 2.1). The Manager App allows approved persons to view the status of the SMADS system, fulfill orders, and command the robot to return to Anna Hiss Gym, the robot depot. Further details on the Manager App and its development are presented in chapter 5.

2.2 Servers

As Figure 2.1 shows, SMADS utilizes a two-sever communication structure. As shown in Figure 2.1 marker 3, the main server, called the App Server, is responsible for managing customer orders and scheduling robot deliveries. The App Server is the central hub for communicating and coordinating information between the customer, via the Texas Botler App, and delivery robot via the robot server. Further details on the App Server and its development are discussed in chapter 3.

The second server component in the SMADS two-server structure is the robot server, which is replicated on each robot in the SMADS system. The robot server is represented in Figure 2.1 by marker 4. The robot server is responsible for translating information sent by the App Server to the robot autonomy stack, shown in Figure 2.1 by marker 5. The robot autonomy stack operates using the Robot Operating System (ROS). Therefore, a main function of the robot server is to convert information sent by the App Server to ROS commands understandable to the robot autonomy stack. Similarly, the robot server translates information generated by the robot autonomy stack to JSON payloads to communicate with the App Server. Status updates, including the current GPS latitude and longitude are sent over HTTPS to the App Server from the robot server using this JSON format.

The two-server structure allows SMADS to share information from an outside network to the SMADS VPN network. This task is performed by the App Server. With a port open to the wider internet, the App Server is able to process requests from the Texas Botler App. Furthermore, since the App Server is hosted on a VPN-trusted UT Austin machine, the server is able to forward information to the robot server over the VPN network. This VPN network is used in part to secure the robots and prevent outside, malicious requests. The two-server system maintains the integrity of the VPN security while allowing communication with machines outside of the VPN network.

2.3 Robot Autonomy Stack

Hosted on each robot, the robot autonomy stack is responsible for processing a destination goal and navigating the robot to that location. As Figure 2.1 shows, the robot autonomy stack is comprised of a robot-level system architecture that coordinates information flow between the various stack layers, namely the localization, navigation and motor controls. The localization software is responsible for reading in LIDAR data gathered from the Velodyne Puck and mapping detected environment features to a known map of UT campus. In the SMADS system, an Episodic Non-Markov Localization (EnML) algorithm was implemented. EnML reasons about the current position from observed observations arising from permanent,

temporary, or moving objects [6]. Once the robot understands its current location relative to a map, the robot attempts to navigate to its destination. The navigation software is responsible for making a global plan to arrive at the destination with small deviations allowed for real-time obstacle avoidance. The navigation stack implemented in SMADS has been previously shown to perform well in long-term deployments of autonomous robots [7].

FIGURE 2.2. SMADS Robot Platforms

2.4 Robot Platforms

The SMADS lemonade stand service was deployed and tested for the week of November 16-20th, 2020, the results of which are discussed in chapter 8. This deployment was carried out using a Clearpath Jackal and a Clearpath Husky (shown in Figure 2.2). Both are four wheeled platforms, equipped with a Velodyne VLP-16 3D Lidar and a pair of RGB cameras. These robots support differential drive steering, and used on board Wi-Fi to communicate over the VPN network. Manual intervention commands are sent to these robots using a PS4 controller. Both operate on batteries with limited lifetimes and extended recharge times. Therefore, to operate the SMADS lemonade stand continuously over a week's time span, several replacement batteries were used to exchange depleted ones, while the depleted ones were being recharged.

Given the general overview of the SMADS system, the following chapters details the mobile applications and server systems used to facilitate communication between the robot and, manager and customer. For more information regarding the other SMADS subsystems, such as the robot autonomy stack, refer to the suggested reading [6], [7].

CHAPTER 3

Application Server Development

As described in chapter 2, this project's overall architecture contains two types of servers: the Application Server (App Server) and the robot servers. Contrary to the robot servers which live and run on each active robot operating in the SMADS system, there is only one Application Server. The following sections detail the App Server's development, the technologies used, as well as its overall architecture.

3.1 Technologies

One of the main focuses during the initial phase of development was to select the right technologies to accomplish our task. Since development began from scratch, any technology from the industry standard to the most recent technologies was viable. Our development time constraint—three months from conception to first tests—encouraged us to choose well-known technologies used in industry, as there is more documentation available for these technologies and a more developed community of developers who have most likely encountered and solved issues we would run in.

Our final choice of technologies is as follows:

Java — Even though Java is not the most modern or popular programming language available, it remains the highly used in the industry [8]. Many well known technology companies, like Google, Uber and Twitter, rely on Java to develop their servers [9]. In that sense, Java is a popular and reliable language in the application server industry.

A server needs to be deployed and run on a computer connected to the Internet to make the business logic it serves available to the public. Java solves this problem since it is platform agnostic. Furthermore, many cloud infrastructures, like Heroku[1] or Amazon Web Services[2], are compatible with Java with very little configuration needed.

Another consideration for the server technology was the available tooling to ease code development . IntelliJ, a well-known Java Development IDE (Integrated Development Environment) was an easy choice to make for our Java server development[3]. Another, just as easy choice, was the industry-standard Spring Boot framework which provides developer with low-level building blocks and streamlines web server development.

Spring Boot — Most application servers are developed on top of an application server framework, which is responsible for the lower level technical logic related to HTTP requests, connection management, etc [8]. This allows developers to focus solely on developing the features they need and increases their productivity and development speed.

Spring Boot was chosen because it is an industry standard [10]. After a short learning curve, the development team was able to move forward at full speed with development. In fact, the initial setup and the first API endpoint development took less than 30 minutes when nominally, without a framework, this development could take hours.

Spring Boot manages the life cycle of all of the application's components automatically. As such, some aspects of the framework's internal logic were treated as a black box. For example, reading and writing to the MySQL database was handled almost entirely by Spring Boot's internal logic—developers only needed to write a SQL query, the rest (database connection, query interpretation and validation, object parsing, ...) was managed by Spring Boot. Similar black box behavior was implemented in our project management tool: Maven.

Maven — Apache Maven, made available by the Apache foundation, the world's largest Open-Source foundation, is a software project management and comprehension tool [11]. In

[1]https://www.heroku.com
[2]https://aws.amazon.com
[3]https://www.jetbrains.com/idea/

short, Maven provides developers with an easy way to manage project parameters, build steps, and most importantly dependencies. Dependencies are sets of code written by third parties and made available to the public to expedite development. Maven is also used in this project to start the server and monitor its health.

MySQL and PostgreSQL — Most web applications rely on a Database Management System (DBMS) to save and query their data [12]. Two of the most popular DBMS's are MySQL and PostgreSQL [8]. They provide an easy way to save and query structured data.

PostgreSQL and MySQL were both implemented in the App Server. The cloud version of the server running on Heroku operated with PostgreSQL. However, local testing on a developer's machine and the deployment server located on a UT machine both operated using MySQL. The motivation for this dual DBMS implementation includes the limitations of Heroku's free services to support PostgreSQL and not MySQL. The bridge software design pattern [13] was employed to select which DMBS to connect to given the which version of the server is running. This choice is made on start up of the App Server and remains unchanged afterwards.

3.2 Backend Representational State Transfer Application Programming Interface

As the name suggests, the main role of an application server is to serve data to clients (websites, mobile applications, desktop applications, etc.). In the case of SMADS, the backend communicated with the customer-facing application, Texas Botler, and the Manager App developed in parallel (see chapter 2 above, chapter 4 and chapter 5 below).

Clients send requests to the server, and expect responses with useful data that the server has processed. The four ways in which a server can manipulate data are Create, Read, Update, Delete (also known as the CRUD operations). For example, Texas Botler can ask the App Server for a list of service locations which are the places the robot is able to deliver to (Read). The manager application can add or delete a robot in the system (Create and Delete). And each robot can update their respective location (Update).

These functionalities are exposed to the Internet through a REST API (Representational State Transfer Application Programming Interface [14]). The anatomy of a rest API is as follows:

```
METHOD /path/to/the/resource
```

The `METHOD` can be one of the four following HTTP verbs, each one corresponding to a specific type of action:

- `GET` is used to retrieve data. Ideally, the logic executed during a `GET` call does not change the state of the server or content of the database;
- `POST` is used to create data or change the state of the server, such as the mode the server is running in;
- `PUT` is used to update an existing record in the database.
- `DELETE` is used to delete a record from the database. However, in most applications, for tracing purposes, records are not actually deleted from the database. They are instead often marked as deleted or deactivated. In the SMADS design, delete requests change the `isActive` field of a record to false;

The `/path/to/the/resource` is a way to represent a resource in the application. For example, the following request

```
GET /serviceLocations/1/latitude
```

retrieves the latitude of a service location identified by 1. (The concept of resource identification is linked to the database which manages the data for the application server and assigns a unique identifier.) And this request

```
POST /trips
```

would create a new trip in the database. `POST` and `PUT` requests are usually accompanied with a "request body" in the JSON format (the JavaScript Object Notation represents data in a lightweight key-value pair structure) which contains additional data relevant to the request at hand.

Since the server contains sensitive data and communicates information to valuable robots, securing the server from outside requests is of utmost importance. In order to guarantee the security of our users' data, all communication between the clients and the server is performed using the HTTPS protocol. Further details on data security in the App Server are presented in chapter 6.)

3.3 Code Architecture

Keeping a code base clean and well structured is no easy feat. In an effort to achieve this goal, the code follows the industry standard MVC (Model View Controller) architectural pattern. This pattern is followed by most actors in the software industry and enables separating concerns [15].

Following this pattern allows developers to implement a layered application which follows the separation of concerns principle. This principle states that each section of a software code base should focus on one area, and one area only. Figure 3.1 presents the different layers implemented in the SMADS back-end.

3.3.1 Models

The Models represent the entities manipulated by the business logic which is code specific to the server's application. For SMADS, business logic would include assigning customer orders to trips, manipulating both the robot and trip models. Other examples of such models in the backend code base are `User`, `ServiceLocations` and `Waypoint`. Models are divided into three categories:

- Domain Models: these models and their corresponding classes represent tables in the database. Java and the Spring Boot framework convert database objects into Java objects automatically;
- Requests Models: these models and their corresponding classes represent the objects sent to the server by the clients during `POST`and `PUT` requests;

- Responses Models: these models and their corresponding classes represent the objects sent from the server to the client after the business logic was executed.

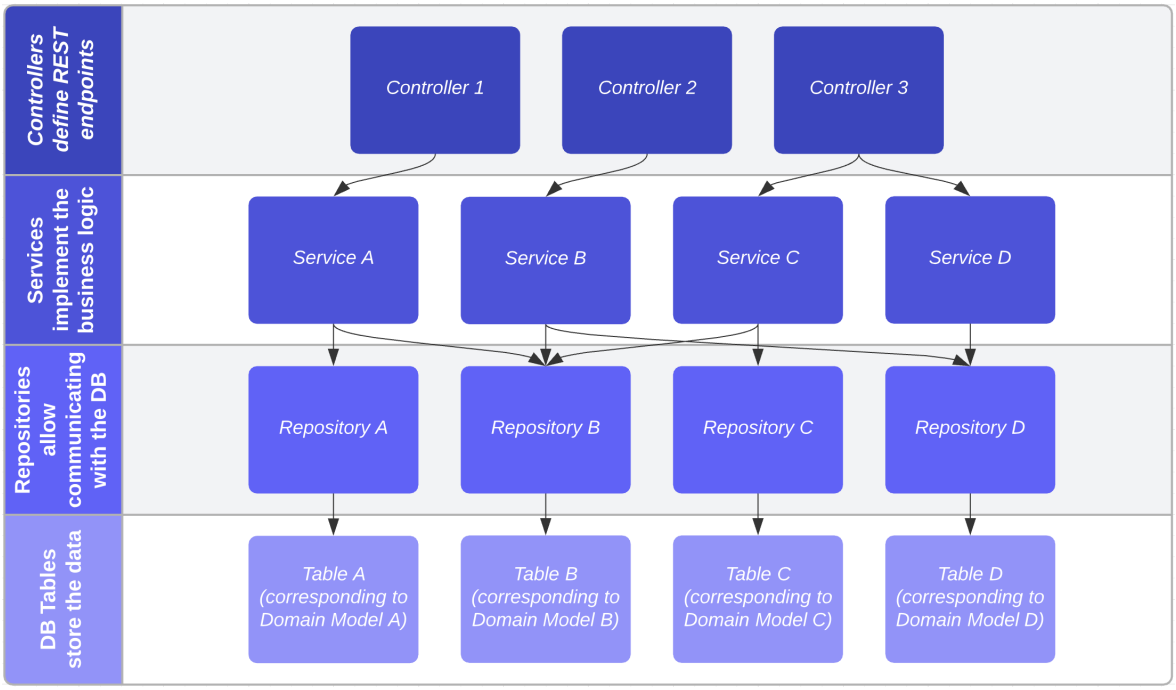


FIGURE 3.1. Overview of the code organization, with Controllers, Services, Repositories, and Database Tables mapped to Models

For both the Request and Response models, the Spring Boot framework takes care of marshalling (converting) the JSON objects to Java objects and vice-versa.

3.3.2 Views

Views are code artifacts that represent data. In any application that contains a User Interface (UI), the views are what the physical users would see. In the context of a server, the REST API endpoints can be considered the views as they allow users, *i.e.* other software like client applications, to see the data they need in a given context.

3.3.3 Controllers

In a well structured and well written back-end application, controllers connect REST endpoints to the business logic needed to be executed to process the request. Controllers should not implement the business logic, but rather call specific services containing that logic. They can however orchestrate and connect different services together and enhance the modularity of the code.

Controllers should group REST API endpoints pertaining to the same topics. For example, the `UserController` should only contain endpoints focused on Users, like `updateUsername`.

3.3.4 Repositories

Repositories are objects that retrieve data. They implement logic to query the database and Create, Read, Update or Delete data (the CRUD operations mentioned above).

Repositories should only be called by services to respect the separation of concerns principle. However, they need not be called by their specific service. For example, the `UserService` can call both the `UserRepository` and the `TripRepository`.

3.4 Communicating with the Robots

Historically, applications were designed in huge monoliths, where every feature was part of a central application [16]. This approach has multiple flaws, including issues such as scalability (if one feature is more popular than most others, that specific feature could not be scaled independently), size of the code base, responsiveness, and robustness (if one feature crashes, the entire system could go down). To avoid such issues and design the platform using a more modern approach, the SMADS system is designed with micro-services, where each robot runs its own robot server. This allows each robot to be independent, improving robustness of the SMADS system. With this design, if one robot goes offline, other may still operate and

communicate with the App Server via their own robot server. This overall design follows a more modern approach to software design: micro-services.

Mirco-services are the contrary to the traditional monoliths of code. In this approach, services are separated into several subsystems that communicate across machines to complete at task. Each subsystem is responsible for a small subset of for alike tasks. The SMADS architecture implements micro-services using independent robot servers which live on each robot and handle the task of communicating information from the App Server to the robot. These robot servers are responsible for this single task. Furthermore, the separation of the App Server from the robot server follows the principle of mirco-services. The opposite approach, the monolith design, would have integrated the robot server's tasks of translating JSON information to robot commands into the App Server. This approach, however, is less robust and more difficult to test and debug. Therefore, communication with the robots occurs over a two-sever structure to enforce separation of concerns and improve the robustness of the SMADS system [17].

When a manager adds a robot to the system, they enter the robot's IP address among other information (see chapter 5 for more details). This IP address is static within the robot's VPN network and thus identifies the robot. Furthermore, this IP address allows the App Server to send information, such as trip details, to the robot server located at that address.

To centralize all communication with the robots in one place, the team developed a `RobotClient` which regroups all necessary endpoints on the App Server. This pattern can be compared to the repository design pattern described in subsection 3.3.4 above. When called, the robot client is passed the robot object which contains the unique IP address of the robot. It thus knows the IP address to which to send the JSON payload to.

Messages sent to the robot include the trip object which contains the delivery destination. Upon receiving this message from the App Server, the robot processes the destination goal and starts the trip.

The server also sends messages to cancel a trip, send the robot "home", *i.e.* to a charging location, and update the robot status (`enroute`, `pickup`, `dropoff`, `charging`, `available`, `outofservice`, `returninghome`, `atHome`, `reconnectingToInternet`,

`assignedTrip`). These robot statuses allow the server to keep track of the state of each robot. The robot state is used mainly to determine which robot to assign to a trip and keep track of moving robots.

Robots communicate their status with the App Server every second. They send an object referred to as the `NewSpotConditionRequest` which contains the robot's status as well as its location, heading and charge level. The client pulls the robot's location to update the position of the robot icon on the map displayed in Texas Botler (see section 3.5 below).

3.5 Dynamic Updates: Sharing updates with the customers

User experience is a key factor in a successful application. Users should not have to manually refresh the application to get the latest status. Instead, the application should update automatically.

3.5.1 Underlying technologies

Communication between client and server can be achieved in two different ways.

The first one and most popular is an HTTP/HTTPS request to a REST API endpoint [18]. These requests are initiated by the client and sent to the server to process. The server cannot send a request to the client using this method.

HTTP/HTTPS requests are reliable and can easily be processed by the server. The corresponding responses are also easily usable on the client. They contain a HTTP response code (each code has a specific meaning) and the useful payload generated by the server and converted to JSON using the Response models described in subsection 3.3.1 above.

The second technology available to communicate between client and server is Websockets. Websockets allow two way communication. This means servers can be proactive and send

clients updates when necessary, without the client needing to initiate the exchange. Websockets connections are initiated by the client, but after the first client/server handshake, the communication pipeline is established and can be used by both parties.

The main use case for sharing updates with the client was updating the robot's location on the map, in a similar fashion as Uber updates the location of your driver before they arrive to the designated pick-up spot[4]. The initial implementation of this feature was using Websockets. Every time the robot sent the server its updated condition, the location was pushed to the client to inform the customer of the progress of their order. However, after testing this implementation over the course of two weeks, it was noticed that the updates were not reliable. A deeper investigation showed that the messages sent through the websocket from the server to the client were never received. This reliability issue encouraged the development team to replace websocket communication with server pulling.

Server pulling is an implementation in which the client makes repeated HTTP/HTTPS calls to the server at a given interval for information. As a result, when the robot send a `PUT` request to update its status, the client's repeated pulling with a `GET` request eventually retrieves the updated information. Information is communicated to the client within a time span of the given pulling interval. For the Texas Bolter app, pulling was set to every 5 seconds. The resulting implementation is more reliable and delivers an similar user experience to the SMADS customers.

3.5.2 Apple Push Notification Service

To avoid forcing the users to monitor their phone continually, the SMADS back-end integrated the Apple Push Notification Service (APNS). Anyone with an Apple Developer Account can integrate their application with APNS. The process requires significant work to set up, but the resulting improvement in user experience is worth the few hours of work.

When a customer opens the Texas Botler App for the first time, they are prompted to opt into receiving push notifications on their phone. If the customer accepts, iOS generates a token in

[4]https://www.uber.com

the form of a string of characters. This token is known as a "push token" and allows Apple and the APNS to identify a specific application on a specific device. If a customer were to sign in to the Texas Botler App with the same account on two different devices, the system would generate two different push tokens.

Each push token is subsequently sent to the server and saved in the database after being linked to the corresponding user. When the App Server decides to send a push notification, for example to inform a customer their order has arrived, the server finds the user's push token(s) and sends it along with the content of the push notification to the APNS servers which deliver the notification the the user's device.

CHAPTER 4

Customer Application Development

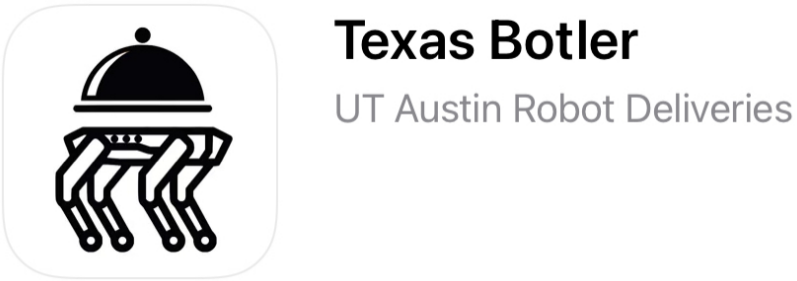


FIGURE 4.1. Texas Botler App Icon

A mobile application is a convenient and effective way to connect with human users in today's world of mobile smart phones. With SMADS' goal of running a lemonade stand, a mobile application is a natural interface to allow customers to place orders. The initial decision to choose to develop an application for iOS instead of Android was mainly linked to the team's expertise at the beginning of the project.

Development started in Apple's IDE, Xcode, which buils and distributes applications for iOS. Xcode has a convenient feature, named Storyboards, which allows developers to visually layout the user interface of their application. Storyboards are an efficient way to visualize the flows of screens in an application. Figure 4.2 represents the Storyboard for the Texas Botler Application.

Texas Botler was developed in Swift 5.1 and UIKit, an established framework to develop User Interfaces for iOS applications.

Texas Botler was developed to be the only interface a customer would ever need to communicate with the SMADS system. The following sections describes the flows contained in the

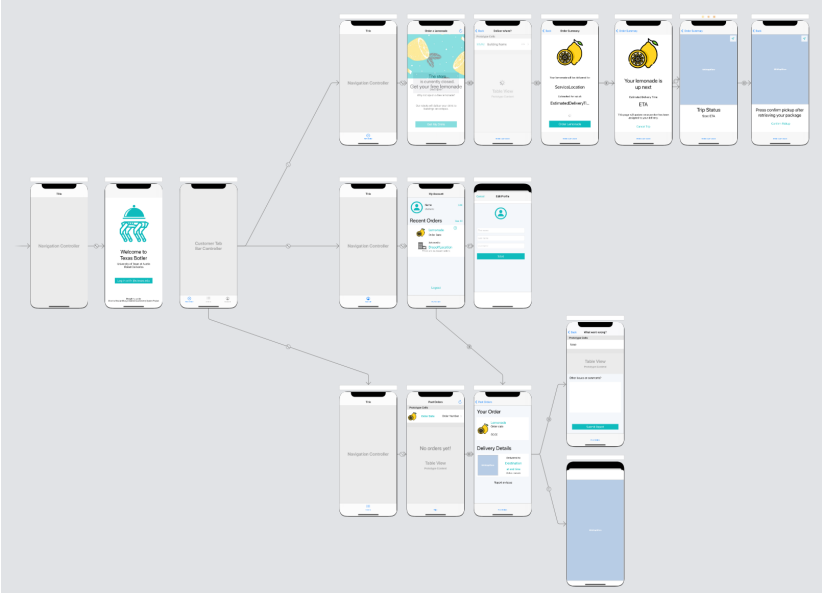


FIGURE 4.2. Storyboard for the Texas Botler Application

application, namely the Login flow (see **??**), the Ordering flow (see section 4.2), the Order History flow (see section 4.3), and the User Account flow (see section 4.4).

4.1 Login Flow

Below is the Login flow for Texas Botler. This set of screens guides the user through the authentication process.

As all of the Texas Botler users are required to hold a university email ending in `@utexas.edu`, and those email addresses are managed by Google, Texas Botler integrates with the Google Authentication system. See Figure 4.3 and chapter 6 for more details on how Google Authentication works.

Once the user has entered their `@utexas.edu` credentials, those are sent to the Google servers for validation. If the requests succeeds, a token is returned to the iOS app. Texas Botler then sends the token to the backend to authenticate the user in the SMADS system. The SMADS server forwards the same token to the Google servers for a new validation. If this second validation succeeds, the response contains information about the user stored in

Google's servers. Some of this information is then saved to the SMADS database for further use.

Once Google validates the token sent by Texas Botler, the SMADS server generates a JSON Web Token (JWT) [19] containing the user's identity within the SMADS environment. JWT is the most secure modern identification mechanism [20]. The token is signed by the server that generates it to ensure its payload cannot be tampered with. The JWT token is subsequently sent by the client to the server with each request to ensure the request has been authorized. It is also used for identification purposes as it contains the user's ID in the database (see section 3.2).

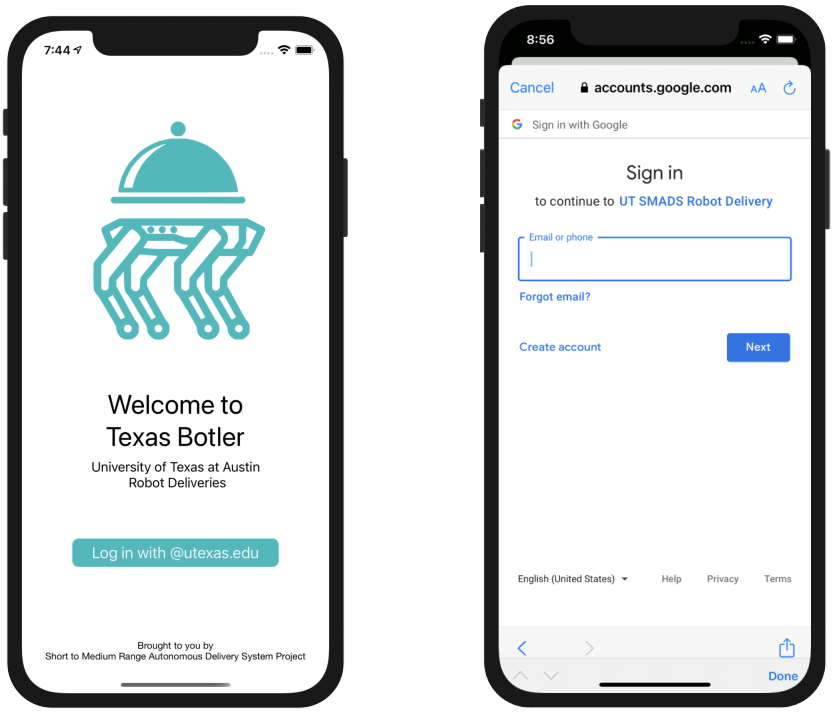
Once the JWT is generated, it is sent back to the client, along with the customer's current order if it exists. This allows the Texas Botler application to present the right view. The token is valid for 7 days, which allows the customer to open the application without having to re-authenticate via Google login every time. This feature is referred to as the *Remember-me Login*.

4.2 Ordering Flow

Upon entering the Texas Botler App, the system automatically selects the first of three tabs. The first one allows the customer to order a lemonade.

The ordering flow is split up into seven different screens. These screens guide the user from the initial step of starting an order all the way to confirming they have successfully picked up the order. Screenshots for for these seven screens can be found in Figure 4.4 and Figure 4.5.

The customer's journey starts with Figure 4.4a where the user elects to start an order. Tapping the "Get My Drink" button leads the customer to a list of service locations (Figure 4.4b). As this screen loads, an HTTP call is made to the backend to load the list of service locations along with their name, acronym, and an estimated time of arrival (ETA) of a hypothetical order if it were to be placed at that moment. All of the service locations in the system are located on UT campus.



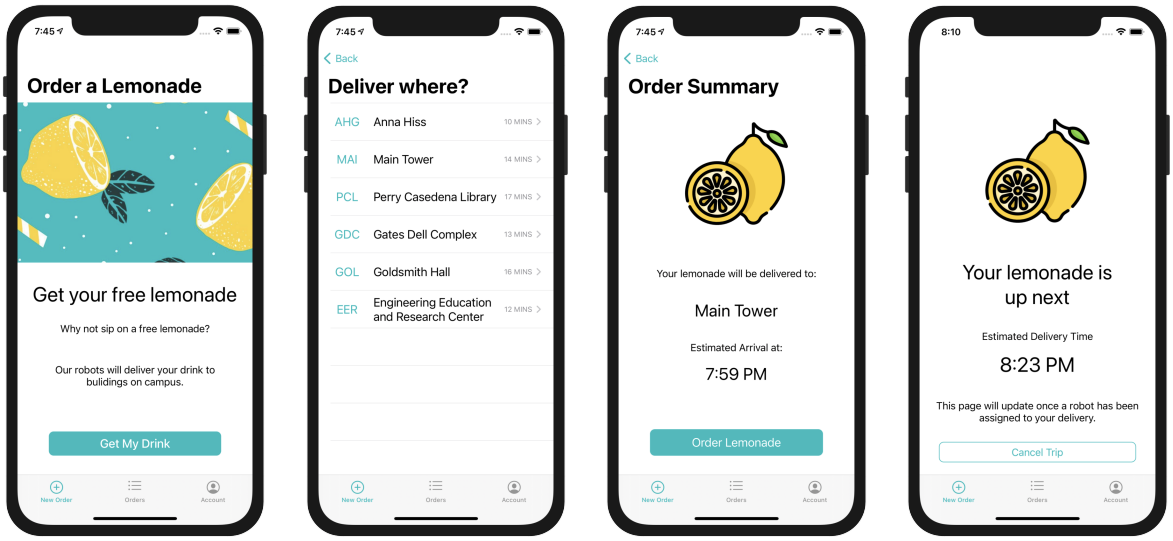
(A) Login Screen

(B) Google Authentication for Texas Botler users

FIGURE 4.3. Texas Botler's Login flow

The user can select their desired drop-off location, which leads them to the order summary screen (Figure 4.4c). This screen displays a summary of the order, including the destination and the ETA. Up to this point, no intent of ordering has been communicated to the server. When the user taps "Order Lemonade", a `POST` call is made to the server to process the order. The business logic behind this process is described in further detail in chapter 7. The trip is then queued from the user's perspective and the Queued Trip screen is displayed (Figure 4.4d). A customer can decide to cancel an order until the robot leaves the robot depot to service the delivery.

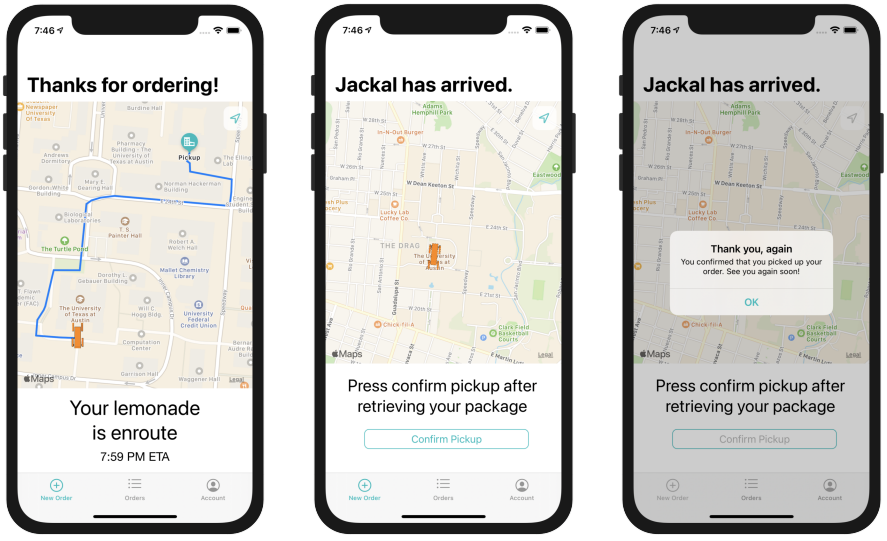
Once the SMADS manager sends the assigned robot on the trip, the robot's route is displayed on a map along with the robot's current position, and the pick-up and drop-off locations (see Figure 4.5a). The robot's location is updated every second using the pulling mechanism described in subsection 3.5.1. When the robot determines it has arrived to the correct drop-off location, it sends a message to the server, which is then transmitted to the client. Upon reception of the arrival message, Texas Botler moves to the Confirm Pick Up screen



(A) Order Lemonade screen (B) List of Service Locations (C) Order Summary screen (D) Queued Trip screen

FIGURE 4.4. Ordering sequence

(Figure 4.5b) and receives a push notification (as described in subsection 3.5.2). When ready, the user can confirm they have picked-up their order. The Texas Botler App then displays a pop-up alert confirming the process is complete.

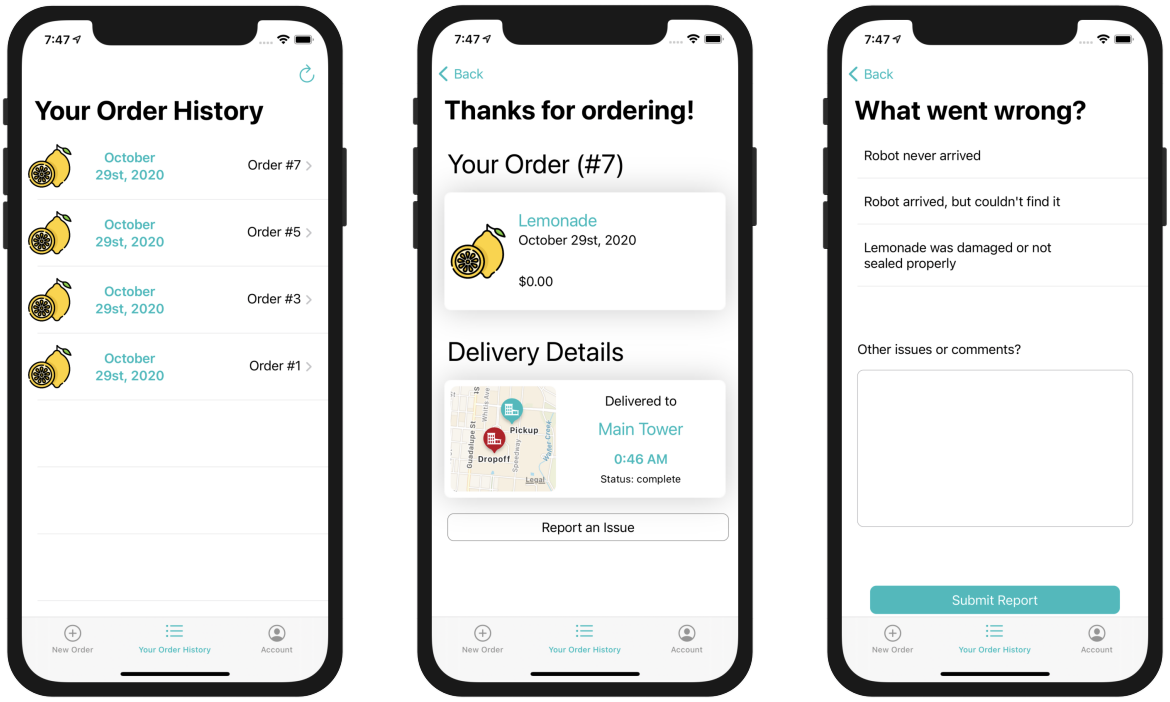


(A) Active Trip screen (B) Confirm Pick Up screen (C) Confirm Pick Up Alert screen

FIGURE 4.5. Ordering flow

4.3 Order History Flow

The second tab provides the customer with an interface to browse their order history (Figure 4.6a). By tapping on an order, the user can see the order details, including the date of the order, the delivery details and a button to "Report an Issue" which they can use to share feedback on their order and their overall experience (Figure 4.6c).

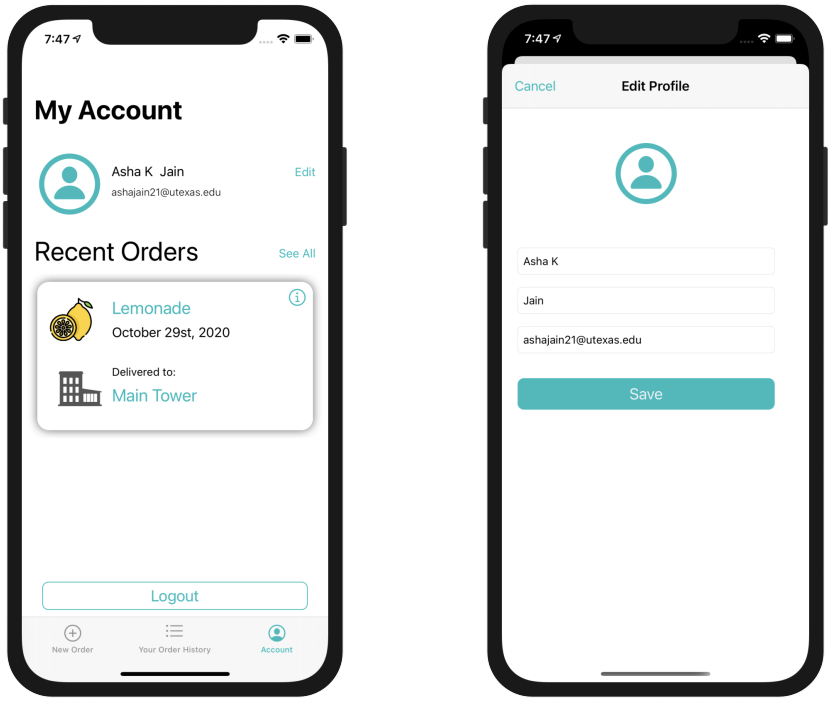


(A) Order History screen (B) Order Details screen (C) Report an Issue screen

FIGURE 4.6. Order History flow

4.4 User Account Flow

The last tab (Figure 4.7a) allows the customer to see their account information and shows the customer their most recent order. The "See All" button links to the second tab to show a list of all past orders. The "Edit" button allows the customer to edit their profile (Figure 4.7b).



(A) Account screen (B) User Profile screen

FIGURE 4.7. User Account flow

4.5 Publishing the Texas Bolter App to the Apple App Store

In order for customers to be able to use the Texas Botler App, the app needed to be made available on the Apple App Store. Apple provides developers with strict guidelines to follow for their application to be approved and published on the Apple App Store [1].

After studying these guidelines closely, the Texas Botler App was built for release and submitted to the App Store review team. A couple days later, the app was available for download. As the team was still testing, we found and solved a few bugs in the application's key features. This required submitting a second version to the App Store.

Apple asks development teams to provide a test account allowing the Apple review team to test the app. This requirement proved more complicated than simply creating a set of credentials for Apple.

[1] https://developer.apple.com/app-store/review/guidelines/

Apple performs tests without notifying developers. The operational requirements of SMADS queues new trips until a manager approves the trip and send off the robot. However, since developers are not aware when Apple's testing team is active, robot managers would likely be unable to process a trip, allowing the Apple testing team to access the active trip flow shown in Figure 4.5. Furthermore, the Apple team would only have been able to place a single order, since the App Server would block further orders until the first order reached completion. Motivated by these limitations, a special version of the App Server was created. A mode, `appleMode`, was created on the App Server to accommodate Apple's testing requirements. When `appleMode` is enabled, orders are directly processed, bypassing the robot managers, and robot movement is simulated so the real systems need not be online at all hours.

The current version of the Texas Botler Application is v2.4.

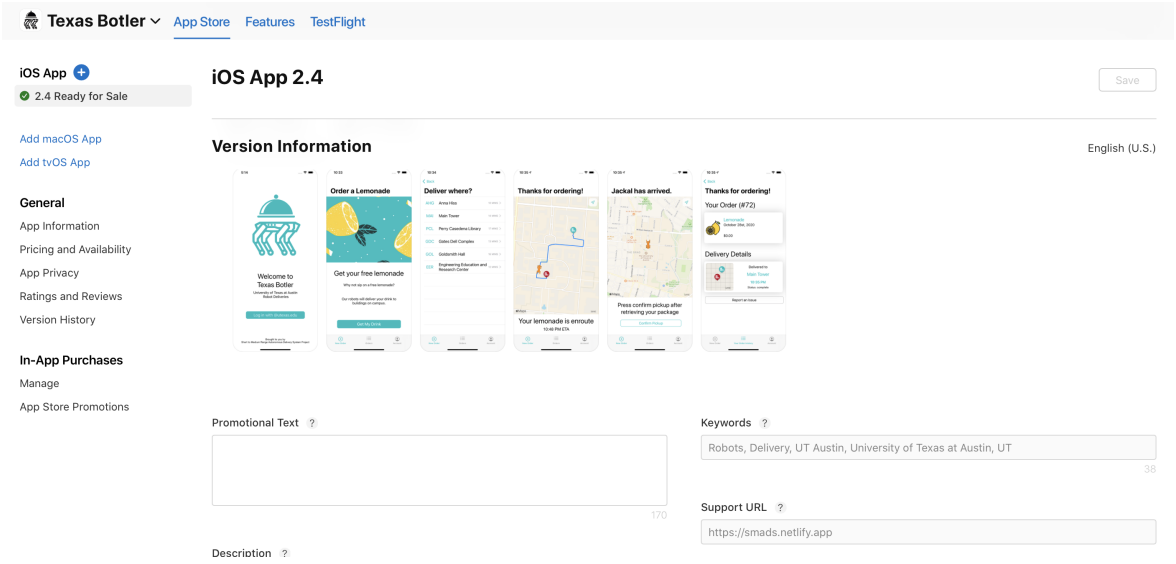


FIGURE 4.8. App Store Connect - Apple's interface to publish Apps to the App Store

CHAPTER 5

Manager Application Development

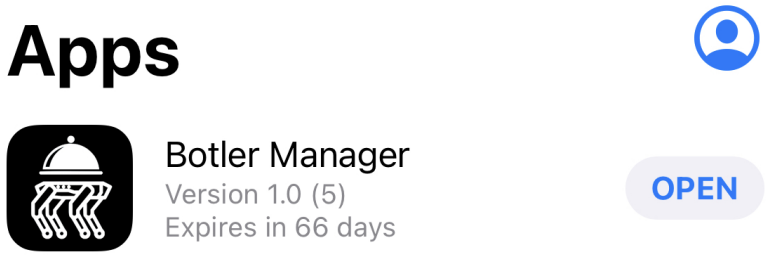


FIGURE 5.1. Texas Botler Manager available through TestFlight, Apple's beta testing platform

5.1 Motivation

Though developing a mobile application for SMADS customers, the Texas Botler App, was a certainty from the beginning of the project, it was unclear at first how managers would interact with the system. The development team initially opted for simple API endpoints which could be called manually using the same tools used during development, namely Insomnia[1]. However, as the project grew in size and complexity — there are 48 endpoints total at the time this thesis is written (see Appendix A) — it became harder to manage interacting with the system. Moreover, the Insomnia interface would only have been usable by a manager who wrote the software; it would have been too technical and intricate to be used by a wider, non-technical audience.

[1] https://insomnia.rest

As such, it appeared logical to develop an interface for SMADS managers to use. The next step was to determine the best kind of client to create. The team debated between a web application, an iPad application and an iPhone application.

Given the project's time constraint, the skills on the team, and the existing code developed for the Texas Botler App, it was easy to eliminate the web application, as this option would have utilized little of the existing code for the Texas Botler App. Choosing between iPad and iPhone for the Texas Botler Manager Application was also easy: (i) some of the Texas Botler Application user interface code could be reused without any modifications, (ii) the team did not have access to an iPad, and (iii) despite an iPad offering more screen real estate to the user, carrying an larger device around while following the robot during tests and deliveries was not as easy as carrying an smaller one like an iPhone.

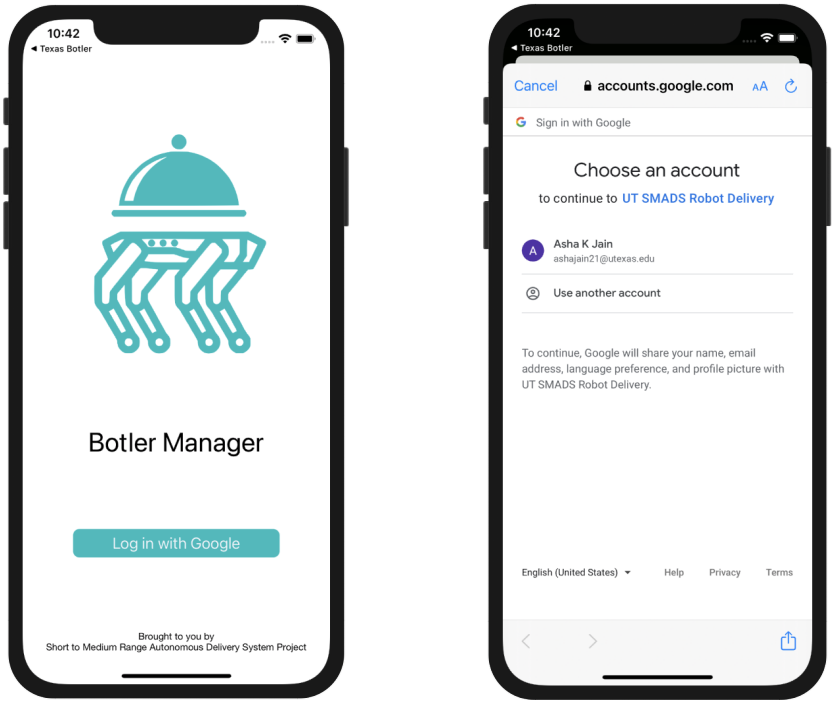
The initial development was expedited by code re-use. On the client, the authentication part of the application, the general design with multiple tabs and the account logic was already implemented. On the server side, a few APIs, including authentication, listing service locations, robots and sending robots on trips were mostly developed. A few modifications were necessary to improve features as development progressed.

The main features the Texas Botler Manager App offers to managers include:

- managing robots;
- managing service locations;
- managing orders;
- managing who has access to the Texas Botler Manager Application; and
- managing their own account.

5.2 Authentication

The authentication process for managers looks very similar to the customer authentication process from a UI perspective (see Figure 5.2).



(A) Home Screen

(B) Google Authentication screen

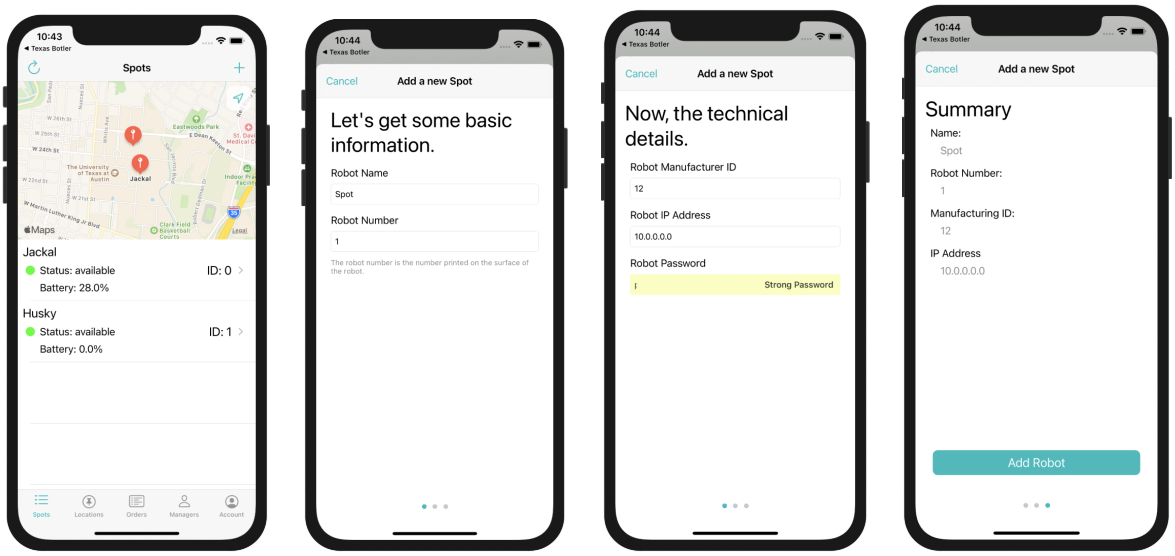
FIGURE 5.2. Manager Authentication flow

After some debate, the decision was made to allow users to be both a customers and a manager. This decision was motivated by the fact that UT-Affiliated individuals have only one `@utexas.edu` email address, and thus, only have one email address which can be used to access SMADS. Since managers needed a way to access the customer app to place orders, managers needed to be able to have two roles, that of a manager and a customer. With this restriction in mind, the authentication API was updated to check if a user has the role of a manager. The Manager App will show an error message if a non-manager tries to log into the Manager App using their `@utexas.edu` credentials (more details in section 5.6 below).

The Texas Botler Manager App implements the same *Remember-me Login* feature as the Texas Botler App, described in section 4.1.

5.3 Robot Management

Figure 5.3 represents the different screens linked to robot management.



(A) List of Robots (B) Add Robot (C) Add Robot technical details (D) Add Robot Summary

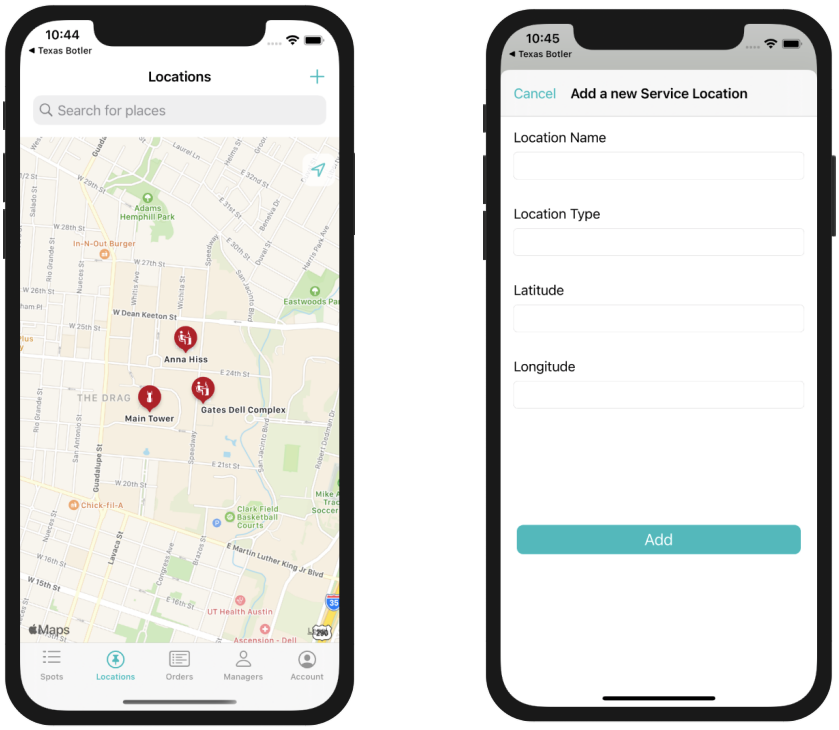
FIGURE 5.3. Robot Management flow

The first page managers are directed to focuses on robot management (Figure 5.3a). Through this interface, they can see a list of all robots registered in the system. Each robot is displayed on the map with more details about the robot's health listed below in a table. These details include the robot battery level, ID and its status. The circular indicator on the left helps visualize the status of the robot. In Figure 5.3a, these indicators are green, indicating the robot is operating normally. They change color to orange if the robot has gone missing (usually due to an Internet connection issue) or red if the robot is offline.

The next three screenshots show screens managers can use to create a robot in the system. Creating a robot requires a robot name and robot number (Figure 5.3b), and the robot's manufacturing ID, its IP address and a password the robot will use to authenticate with the App Server when it starts up (Figure 5.3c). More details about how the robot authenticates with the system can be found in chapter 6.

5.4 Service Locations Management

A manager should also be able to manage service locations which are the places the robot is able to navigate to. This can be done easily by using the second tab.



(A) List of Robots (B) Add Robot

FIGURE 5.4. Service Location Management flow

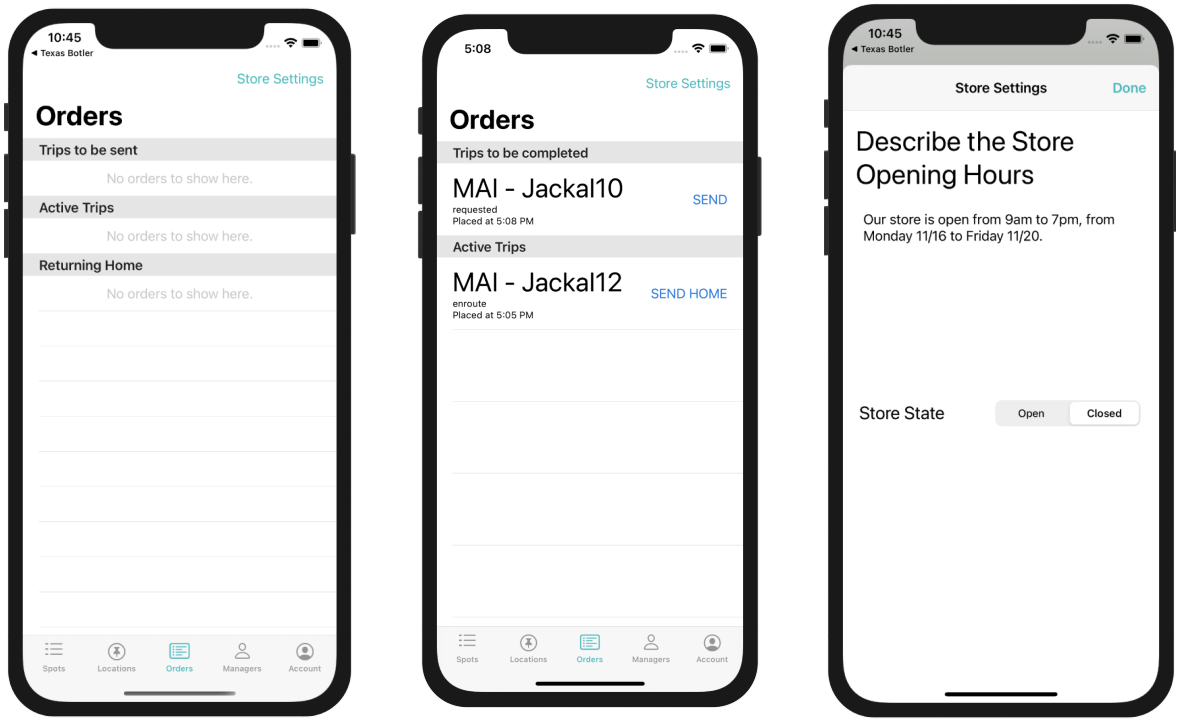
The view loads with a simple and efficient spacial representation of the service locations on the map (Figure 5.4a. Each pin specifies the type of building a location is. The different location types are: `library`, `office building`, `dorm`, `restaurant` and `other`.

A manager can also use the locations tab to add a service location (Figure 5.4b). This can be done by adding a name, location type, latitude and longitude. Future enhancements to this user interface could include limiting the number of options for the location type via a drop down menu, adding a field for the service location's acronym, and using a map view to select the location's latitude and longitude. The development team prioritized the minimum

viable product and focuses on essential features. As a result, the aforementioned features were reserved for future work.

5.5 Orders Management

On top of monitoring the robots during the test phase of the project, managers were responsible for loading the payload, a bottled lemonade during the project's test phase, on the robot and triggering the trip. The user interface for these features can be seen in Figure 5.5 below.



(A) List of Trips (B) Trip Management (C) Store Management

FIGURE 5.5. Order Management flow

When the store initially opens, there are no orders to be serviced. A manager would then see Figure 5.5a. The list contains three sections:

- *Trips to be sent* — displays the orders placed by customers along with the trip's destination, the robot assigned to the trip, the trip's status and the time the order was placed at. On the right, a "SEND" button is visible when the trip has been

assigned a robot. Tapping the "SEND" button sends the trip information to the robot (Figure 5.5b) and begins the delivery. Once the robot is done calculating the route, the planned delivery route information is sent to the Texas Botler App. The customer is then presented with Figure 4.5a and can monitor the status of their delivery.

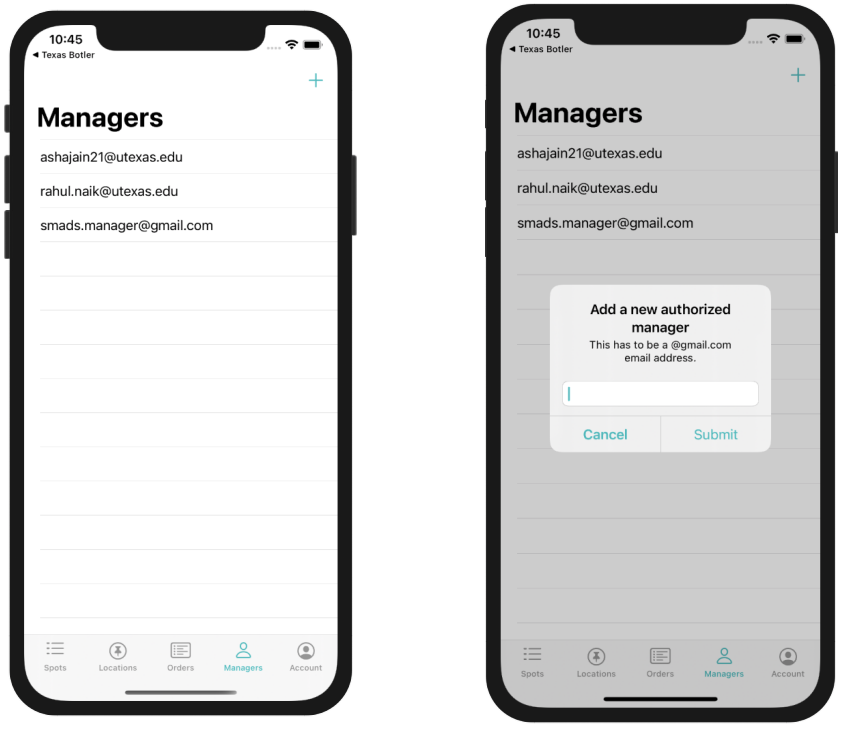
- *Active Trips* — displays the trips and robots currently fulfilling an order, along with the same information displayed in the *Trips to be sent*.
- *Returning Home* — displays the robots on their way back home to Anna Hiss Gym. These trips are usually performed without a payload. A robot is not declared *available* until it has completed the returned trip home.

When a new trip is placed by a customer, the App Server will trigger a process to send a push notification (see subsection 3.5.2) to all managers to inform them of the order. The notification encourages managers to open the Texas Botler Manager App and fulfill the order. This alert provides the managers with a better user experience so they do not have to constantly monitor their device. It also helps the SMADS team provide customers with a better user experience by minimizing the processing time.

To limit the opening hours, a manager can decide to open and close the store as shown on Figure 5.5c. The message entered on this screen is displayed on the Texas Botler App to customers when the store is closed.

5.6 User Management

As mentioned in section 5.1, managers can use the same account, *i.e.* their `@utexas.edu` email address, to log in to both the Texas Botler App and the Texas Botler Manager App. However, to restrict access to the Texas Botler Manager App, existing managers must add the email addresses of new managers to authorize their access to the system. This process is shown below in Figure 5.6.



(A) List of Managers

(B) Add a Manager

FIGURE 5.6. User Management flow

Through this interface (Figure 5.6b), new managers are granted access to the Texas Botler Manager app. Other non-managers who try to authenticate in the Texas Botler Manager App will be presented an error message suggesting they use the Texas Botler App instead of the Manager App. The system ships with a default manager account: `smads.manager@gmail.com`.

5.7 Account Management

In a similar way customers of the Texas Botler App can manage their account, managers can manage their account using the far right tab.

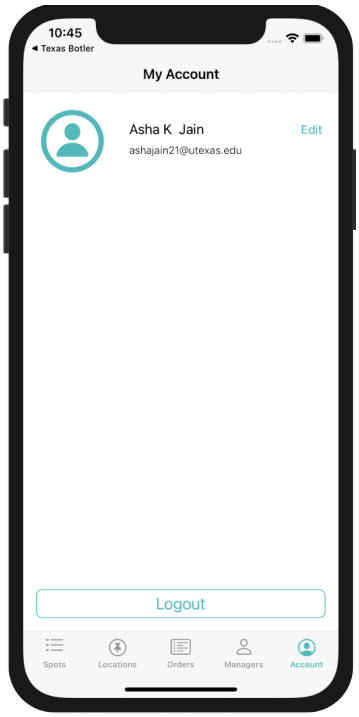


FIGURE 5.7. Manager Account

5.8 Deploying the Texas Botler Manager Application

Once the Texas Botler Manager App was developed and ready to be used, it was time to make it available to managers. Like the Texas Botler App, the Texas Botler Manager App was submitted to the Apple App Store to be published. However, the App Store testing team ruled that the audience for the Texas Botler Manager App was not large enough to make it available through the App Store.

An alternative solution was to use TestFlight, Apple's testing platform [21]. TestFlight allows developers to publish their application and make it available to a limited audience. This process requires sending invitations to each tester containing a code that can be used to redeem and download the app. Since only four UT students volunteered to be managers, this solution was sufficient for this project, and the Texas Botler Manager App was released through TestFlight to the managers.

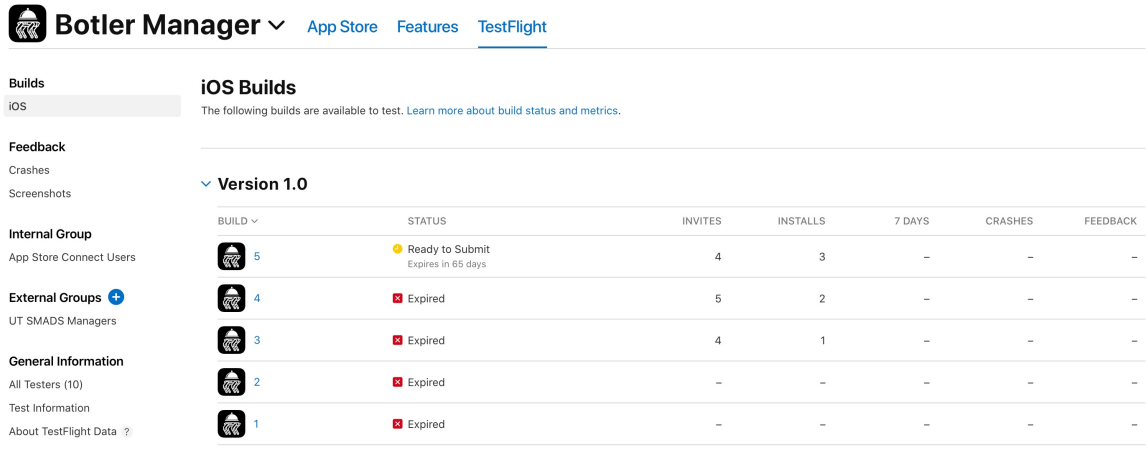


FIGURE 5.8. Texas Botler Manager uploaded to TestFlight

CHAPTER 6

Data Security

Considering this system interacts with personal data over the Internet, securing sensitive information is a central part of the app-server system. Both access to the SMADS system and personal information is restricted.

The SMADS system authenticates users and robots before allowing either to interact with the system. Users, such as customers and managers, authenticate via the Google Authentication client API. Figure 6.1 provides an overview of how this authentication works.

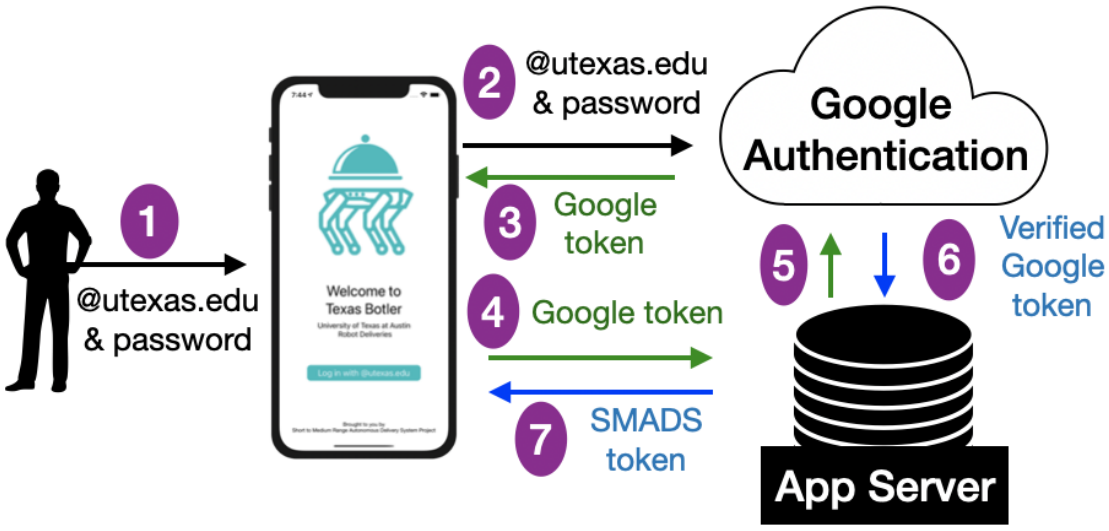


FIGURE 6.1. Customer and Manager Google Authentication

6.1 Customer and Manager Authentication

As seen in Figure 6.1, the customer or manager enters their `@utexas.edu` email credentials into the Texas Bolter App via the login page. The `@utexas.edu` email domain is reserved

for people affiliated with the University of Texas at Austin. Accounts with this email address must first pass UT Austin authentication before creation and activation. As a result, people with an `@utexas.edu` address are verified UT Austin-affiliated persons. The SMADS system leverages this property of `@utexas.edu` accounts to limit access to SMADS to only those affiliated with the university. This restriction helps prevent malicious actors tampering with our system.

The `@utexas.edu` email address is associated with the G-Suite. Thus, authentication of this address occurs through Google Authentication. The customer or manager enters their `@utexas.edu` email address and corresponding email password directly into the Google Authentication graphical user interface (GUI) (see Figure 6.1 for this GUI). Typed passwords are hidden behind secure text and are not displayed in alpha-numeric characters to maintain the password's security.

Once the customer or manager presses the Google Authentication login button, Google servers verify that the provided email and password match a known Google account. It is important to note that Google Authentication does not check if the email address is affiliated with the University of Texas at Austin. It simply checks whether the email address and password combination is registered with Google. The check for the email handle, `@utexas.edu` , occurs in step 5 in Figure 6.1.

If the Google servers are unable to verify the existence of the user's email address and password, the Google GUI prompts the user to re-enter their credentials again. The user has up to three times to enter a correct email and password combination. Beyond this tolerance, the Google GUI will dismiss and the Texas Botler App will return the customer or manager to the main login screen (shown in Figure 4.3).

On the other hand, if the Google authentication system is able to verify the user's credentials, the Google servers return a JSON payload containing user information, and importantly a Google Authentication token. This step is shown in Figure 6.1 as step 3.

The Texas Botler App forwards the provided Google Authentication token to the App Server in a HTTPS request to login in the user (shown in step 4 in Figure 6.1). The App Server independently verifies that the forwarded Google Authentication token is valid. This check prevents malicious third parties from accessing the SMADS system with fake Google Authentication tokens sent to the App Server. In step 5, Google Authentication Client is again called to verify the forwarded Google token. If the token is valid, Google Authentication returns the a JSON payload containing the user's information. Otherwise, the Google token is deemed corrupt and the App Server refuses the request, returning a server code `400 BAD - REQUEST`.

With a valid Google Authentication token verified independently by the App Server, the system begins processing the request to login the user. First, the App Server checks if the user's email contains the email domain `@utexas.edu` or if the user's email is a member of the approve emails list. If neither are true, the request is denied and the App Server returns a null object and sets the server status to `403 - FORBIDDEN`. Otherwise, the system continues processing the request.

Next, the App Server checks if the user's credentials already exist in the database, using the assumption that emails are a unique identifier for a user. This assumption is valid since Google Mail suit does not allow a person to create an account with an email address that already exists. If the user's email is found in the database, the App Server works to login the user, generating a SMADS token using JSON Web Token (JWT) authentication (refer to discussion on JWT in section 4.1). This token not is specific to the machine running the App Server, as long as the signing key is identical. In addition to sending the SMADS token, the App Server packages data regarding the user's role in the SMADS system and if the user has an active order. This data is wrapped in `AuthenticationReponse` object and returned to the Texas Botler App.

If the user's email is not found in the database, the App Server records user information in the database and then returns an `AuthenticationReponse` object with a valid token. The `Users` table in the database stores a user's first and last name and email address (referred to as username).

Sensitive information like user and robot passwords are encrypted using SpringBoot Security which relies on BCrypt, a well-established encryption hashing algorithm [22]. In the database, passwords appear nonsensical. Any attempt to use the passwords as displayed in the database will fail to validate and thus fail to login. BCrypt uses its internal processing to match encrypted passwords to their decrypted alias. Through this internal system, BCrypt is able to determine if a user provided password matches the encrypted password stored in the database.

6.2 Robot Authentication

In addition to customers and managers, robots must also authenticate with the SMADS system. Robots have a programmed username and password. Verified managers can create a robot account in the Texas Botler Manager App, saving these robot credentials into the database (refer to Figure 5.3b). Again, passwords are encrypted, following the encryption method explained above. When a robot becomes online, its robot server sends a login request with its programmed username and password to authenticate with the SMADS system. The App Server verifies that the provided username and password match the known credentials. If this check passes, the App Server returns a SMADS token to the robot server.

All HTTPS requests except login requests require that the requester provides a valid authentication token in the HTTPS header. As a result, the sender of each request is verified to be a known and trusted user or robot. This check prevents malicious third parties from sending and receiving information to and from the SMADS system. If a request fails to provide an authentication token header or provides an invalid token, the request is rejected and the server status is set to `403 - FORBIDDEN`. `JWTFilter`, a custom class design for SMADS, handles rejecting or accepting requests based on the authentication header.

6.3 Role-Specific Endpoints

Additional layers of protection were implemented to shield certain endpoints of the App Server from customers. For instance, only managers, neither customers nor robots, can access

the endpoint to create another robot account. This shielding prevents customers and robots from making unwanted changes to the SMADS system. The App Server employs user roles to accomplish this task. Possible roles include customer, robot, and manager. A manager can be both a customer and a manager. A customer cannot make themselves a manager, as the endpoint to create a new manager is restricted to manager accounts. Thus, only already existing managers can create other managers. A robot cannot have the role of a customer nor a manager.

The Spring Boot framework used to develop the App Server handles securing an endpoint to the correct role. The annotation `@Secured(<Role>)` is placed above the class declaration that contains the shielded endpoints. Accordingly, all endpoints in the secured class have restricted access to users that contain the role specified in the angle brackets.

6.4 Data Security During Transmission

With the so far explained data security measures, data is securely stored in the database, accessible only by verified users with the appropriate role. However, data still must be secured during the transmission of information between the App Server and the apps or the the robot servers. The HTTPS protocol was implemented to accomplish this task.

Different from HTTP, HTTPS is built on top off the established Secure Socket Layer (SSL), using signed certificates to verify the identity of a request [23]. These signed certificates can be generated by well-known certificate authorities or by the local system, in our case the SMADS system. Local certificates are called self-signed certificates. Initially, the SMADS system attempted to use a self-signed certificate to implement secure web communication. However, issues with verifying this self-signed certificate in the Texas Botler App could not be overcome. Instead, a certificate from a certificate authority was obtained and implemented. The third-party verification removed the challenges with verifying a self-signed certificate in the Texas Botler App as the iOS environment is pre-configured to verify certificates from established authorities.

CHAPTER 7

Robot and Trip Management

A central task assigned to the App Server is handling order requests from multiple users and scheduling delivery trips for a fleet of robots. The App Server's robot and trip management decision tree with the various states of the customer, manager and robot are shown in Figure 7.1.

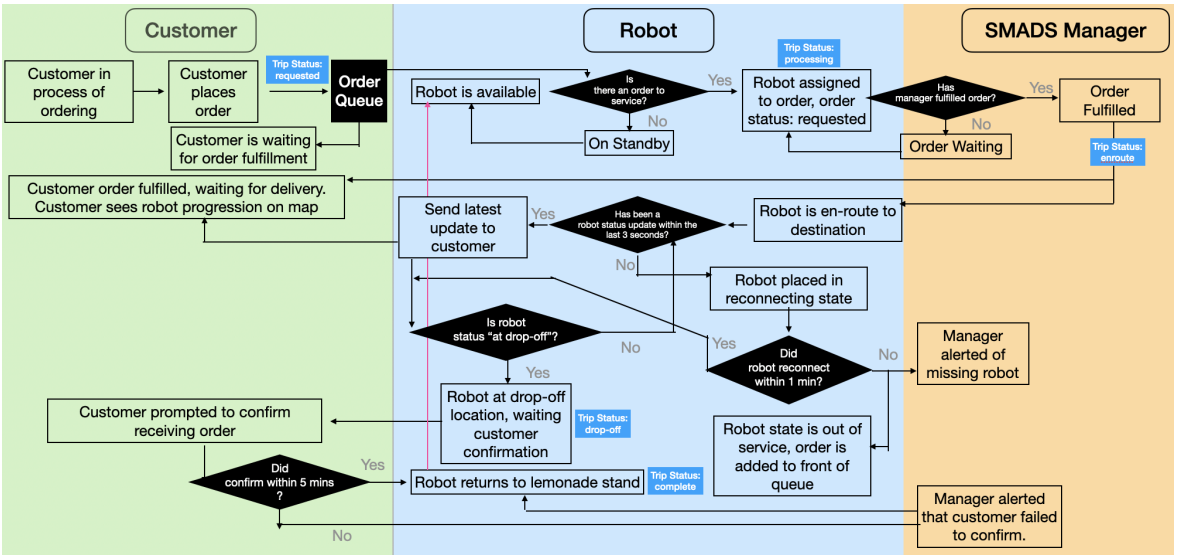


FIGURE 7.1. App Server Decision Tree for Customer Orders and Robot Trips

7.1 Receiving Customer Orders

As shown in Figure 7.1, every order begins with the customer. The Texas Botler app allows users to order a lemonade, selecting a delivery destination from a list of locations. Refer to Figure 4.4 to see the ordering sequence and steps to place an order.

Once the customer places an order, the Texas Botler App sends the order information, such as the selected destination and the calculated eta, to the App Server. A trip object is created with this information, storing the customer ID, destination, ETA and order time. At this point, the trip's status is `requested`.

Now, the App Server must make a decision: should the requested trip be queued or immediately serviced? To service a trip, there must be an available robot. A robot is available if it is located at Anna Hiss Gym, where the robots live and recharge, and the robot has not already been assigned a trip. Since the App Server maintains a record of the current robot state, the server reads stored robot data from the database to see if there is an available robot. The robot status field in the robot table is checked, looking for an robot enum status of `available`. If an available robot is found, then the trip is immediately serviced. The first available robot is assigned to the trip and the trip status changes to processing.

If no available robot is found, the trip is queued. This trip state is represented by the "On Standby" box in Figure 7.1. The database contains a trip table which serves as the trip queue. A query reads in the requested trips ordered by start time. Older orders are listed first. The queue is checked each time a customer orders a trip or the robot status is updated to `available`. Therefore, when a new robot becomes `available`, the queue is reviewed and the oldest request is serviced.

7.2 Serviced Trip Processing

Once the order is serviced, the App Server sends the trip and assigned robot information to the Manager App. A robot manager must fill the customer's order, placing a lemonade on the assigned robot. Furthermore, the robot must be taken outside of Anna Hiss Gym, and localized before it is ready to deliver a robot. The robot manger is responsible for completing these tasks before allowing the robot to start navigating to the destination. Until the manager is able to confirm that these tasks are completed, the order waits and status remains in processing. As seen in Figure 5.5, the manager can use the Manager App to confirm that robot has the

lemonade, is outside, has been localized and is ready to go. The manager presses the "Send" button to confirm this status.

Once the order is fulfilled, the trip status changes to `en-route` and the robot status changes from `assigned to trip` to `en-route`. At this point, the App Server sends the robot the destination details via the `RobotClient` class. The robot processes the destination goal, creating a global navigation plan to reach the target. The App Server retrieves the intermediate way points from the robot's global plan and forwards the information to the Texas Botler App. These waypoints are used to draw the intended delivery route on the customer map's view (shown in Figure 4.5a).

7.3 Active Delivery Robot

As the robot begins navigating to the order's destination, the robot updates its status every second, including its current location in GPS latitude and longitude and battery health. These updates are sent to the App Server over HTTPS communication as described earlier in section 3.4. The App Server records the status updates in the database and forwards the current location to the Texas Botler App to be displayed on a map for the customer. If the App Server does not receive a status update from the robot at least every 3 seconds, the robot is placed in a `reconnecting` state. This reconnecting state handles the case when the robot loses its WiFi connection and can no longer send updates to the App Server. This issue often occurs in outdoor systems since outdoor WiFi connections are typically unreliable. Around UT campus, several dead zones were identified, including around the Main Tower and near the Turtle Pond.

Once the robot is in a `reconnecting` state, the App Server waits a minute to see if the robot will reconnect. Often, as the robot progresses to different parts of campus, the robot is able to reconnect. If the robot does reconnect, it sends a status update which is recorded and forwarded to the Texas Botler App. However, if one minute elapses with no update from the robot, the App Server changes the robot's status to `missing` and notifies the manager of a missing robot via the Manager App in a push notification. At this point, the App Server

assumes that the robot manager will retrieve the robot and reconnect it to the system. All connected robots continuously send their status updates to the App Server, and therefore, the reconnected robot will override the previous missing state with its current state, which is mostly likely available, resetting the system.

The App Server checks each robot status update to see if the new status is `at drop-off`. The robot changes its status when its navigation and localization algorithm has determined that the robot has arrived at its target destination. Once this `drop-off` status is received, the trip status is changed to `at drop-off` and the App Server prompts the Texas Botler App to change screen. The customer now sees the Confirm Pickup screen (shown in Figure 4.5b). The customer confirms that they have retrieve the lemonade from the delivery robot by pressing the button "confirm pickup." This button triggers code that makes an HTTPS call to the App Server to send the robot back to Anna Hiss Gym. A trip object is created to represent the returning-home trip. The App Server handles this trip much like the delivery trip, listening to status updates from the robot to monitor the robot's progress.

7.4 Order Completion at Destination

Customers are not a reliable, and thus the App Server does not depend solely on the customer to send the robot home to Anna Hiss Gym. The App Server waits 5 minutes for the customer to press the confirm pick button before it handles sending the robot home itself. However, in the case that the robot loses WiFi connection at the destination, the App Server will not be able to communicate the command to return home to the robot. As a result, the manager has a back-up method to send the robot home using the Manager App. In the manager app, the trips screen (shown in Figure 5.5) allows the manager to press a button called "SEND HOME" for a certain trip — see Figure 5.5b), prompting the robot to return home. The App Server checks if a robot has already been sent home before creating a new returning home trip.

Once the robot reaches Anna Hiss Gym, the App Server resets the robot status to available and sends a notification to the robot to update its status from returning home to available. As mentioned before, any changes to the robot status prompts checking the trip queue. Therefore,

if there is another order waiting to be serviced, the App Server will read in robot data to see if any robot is available, and likely the newly available robot will be assigned a new trip. This reassignment restarts the decision tree process, back to the state when the robot is assigned to a trip. This loop is highlighted in Figure 7.1 with the pink arrow.

7.5 Handling Multiple Orders

This robot-trip management is capable of handling multiple orders at once using a fleet of one or more robots. If the App Server receives multiple orders at the same time, the App Server will attempt to service as many orders as possible, assigning one order per robot. Thus, the App Server's ability to service trips is limited by the number of robot available in the fleet. The App Server imposes a rule of one order per robot in order to ensure that the customer retrieves their order only. Furthermore, this rule simplifies scheduling.

The process described in Figure 7.1 occurs for every customer order. Furthermore, the App Server maintains the state for every robot, recording the status updates and verifying that the robot is still connected. With this error handling, the App Server is able to handle several robots operating in real world scenarios, which are highly error-prone and introduce unexpected issues, like robots disconnecting from the Internet. Another benefit of this design is that the App Server can assign trips agnostic to the robot platform. As a result, this server system can be applied to wheeled robots or legged ones with no changes to the ordering sequence or scheduler. This feature is useful to future work, especially as other researchers may not have the same robot fleet as used to test this system.

CHAPTER 8

Field Results

The SMADS system was deployed on the Clearpath Jackal and Clearpath Husky (refer to Figure 2.2) for a week of testing from November 16-20th. In the following sections, the experiment set up, successful results and recorded issues are discussed.

8.1 Testing Set Up

Acting as a modified lemonade stand, the SMADS system received orders from customers for lemonades and coordinated robots to deliver the bottled drinks to various destinations. Possible destinations included the Main Tower (MAI) and the Gates Dell Complex (GDC). These destinations had their delivery routes previously mapped and tested on the Clearpath Jackal robot.

Robots waited for lemonade orders at Anna Hiss Gym, a campus building home to a shared robotics space. At all times, two robot managers were present with the autonomous robot. Managers were responsible for logging issues, videoing the robot trips and intervening with manual commands if the robot system was in danger. Manual interventions can occur when unexpected car traffic or crowds of people are placed in the robot's path. Due to limited student availability to serve as robot managers, the SMADS lemonade stand was not active continuously between Nov. 16th and Nov. 20th. Table 8.1 details the number of operational hours that occurred over the course of the testing week.

Nov. 16th	Nov. 17th	Nov. 18th	Nov. 18th	Nov. 20th
9 hrs	7hrs	6hrs	3hrs	6hrs

TABLE 8.1. SMADS Hours of Operation

Managers open or close the SMADS store using the Manager App as discussed in section 5.5. When the SMADS store is close, customers are not able to place orders. This feature ensures that robots are not moving to service orders without the supervision of a robot manager.

The choice of lemonade as the good to deliver was deliberate. Lemonades were bottled drinks to prevent liquid from spilling on the robot. Furthermore, since lemonade is not carbonated, the vibrations from the robot rolling on uneven pavement would not pressurize the drink. These two features made lemonade a suitable drink to deliver.

8.2 Procedure

The procedure for running the SMADS lemonade stand is as follows:

(1) Step 1: A SMADS robot manager logins onto the Manager App. If the manager does not have the app, the manager follows steps outlined in section 5.8 to acquire app.
(2) Step 2: In the Manager App, navigate to the "Spots" tab to view the current state of the SMADS system. If there are no robots listed, proceed to step 3. If robots appear, proceed to step 4.
(3) Step 3: Turn on the robot platforms by pressing the on button. Locate the PS4 controller for each robot. Verify that the robot has appeared in the Manager App once online.
(4) Step 4: Check for an order to fulfill in the Manager App tab labelled "Orders."
(5) Step 5: Once there is an order to fulfill, remove the assigned robot from the charger. Obtain the issue log notebook and the PS4 controller. One robot manager should also carry their laptop in case its necessary to localize the robot.
(6) Step 6: Take the robot outside and localize if necessary. Once localized, press the "Send" button for the order listed in the "Orders" tab in the Manager App.

(7) Step 7: The robot is processing the trip and has created a global plan to navigate to the destination. Enable autonomy mode using the PS4 controller. Begin recording.

(8) Step 8: Managers, be aware of your surroundings and intervene if the robot appears to be headed for danger. Manual commands on the PS4 controller will stop autonomy mode and override the navigation motor commands. Record all interventions. If need be, re-localize the robot after an intervention.

(9) Step 9: Once the robot has arrived at the destination, wait for the customer to arrive. Managers can see the customer name in the Manager App under "Orders." If the customer does not arrive in 5 mins, the robot will begin to autonomously return to Anna Hiss Gym. If the robot is offline, the manager will need to send the robot home using the "Send Home" button in the Manager App under the "Orders" tab. Repeat steps 7-8 until the robot arrives at Anna Hiss Gym.

(10) Step 10: Check the Manager App to see if there is another order waiting. If so, repeat steps 6-9. If not, take the robot inside, place it on the charger and wait for another customer order.

8.3 Successes

Over the course of the week, SMADS successfully completed 27 delivery trips. Five of the 27 trips were ordered to the Gates Dell Complex, while 14 trips were sent to the Main Tower. The remaining trips are the return home trips where the robot autonomously navigates back to Anna Hiss Gym after delivering a lemonade. Nine of the 27 trips were ordered by spontaneous customers who downloaded the Texas Botler app via a flyer. The remainder were orders placed by robot managers.

Trips to each destination were run successfully on both the Clearpath Jackal and Clearpath Husky. The Jackal was able to avoid obstacles, namely a traffic cone placed in its nominal path. Furthermore, the Jackal slowed down when it came close to another object, including curbs and people.

The App Server correctly handled situations of multiple orders placed at similar times. The App Server serviced the first order and assigned it to the first robot available in the database. While that order was en-route, the App Server received notice of another customer order which the server correctly queued. The server calculated an estimated time to deliver to be 52 mins which is a reasonable estimation, given that the only robot in operation had just left to service an order.

The Texas Botler App registered 38 new users, although not all of these users placed an order. The app correctly displayed when the store was open or closed. Furthermore, the app was able to receive order status updates from the App Server and display them to the customer. The queued trip view controller, shown in Figure 4.4d, correctly dismissed and moved to the active trip view controller, shown in Figure 4.5a, when the order status changed to `en-route`. On this screen, the app successfully displayed the current position of the robot and its heading, although issues with Wifi connection on the robot sometimes interrupted this display.

The Texas Botler App correctly moved to the confirm pickup view as shown in Figure 4.5b when the trip status changed to `at dropoff`. This screen successfully allowed the customer to confirm their pick up and dismissed the delivery robot, sending the robot back to Anna Hiss Gym. Furthermore, customers were able to leave feedback on their delivery via the app. Customers reported that the robot did not arrive on time, as predicted by the estimated time of delivery.

The Manager App performed as expected. Managers were able to see placed orders, send robots on trips and return robots home. Furthermore, push notifications successfully alerted managers of offline robots. Managers were able to verify this robot status using the Manager App "Spots" tab. Finally, managers were able to locate the customer using the information displayed in the Manager App for a given order.

8.4 Recorded Issues

Several challenges with the SMADS system were recorded. A total of six issues were recorded over the week-long testing. Of these six issues, five were related to localization. Furthermore, the majority of the issues occurred on the Huksy rather than the Jackal. From preliminary data analysis, the localization issues seem to be rooted in differences in collected sensor data from the Velodyne LIDAR. The Husky mounts the Velodyne sensor in a different location than the Jackal. This difference is enough to change the recorded cloud point data. The navigation and localization maps were generated using the Jackal sensor data, rather than the Husky. It is probable that the sensor data collected on the Husky poorly aligns with the Jackal-derived maps, and thus led to several localization errors. This misalignment would explain why the Husky had the majority of the localization issues.

The one non-localization issue recorded was an error in the Manager App. After processing the data and tracing events, it was discovered that while Jackal was out servicing a trip, the robot had become offline. The robot continued navigating towards its goal, even while offline, and repeatedly tried to reconnect to the Internet. The Manager App correctly showed that the Jackal had lost internet connection and the App Server correctly place the robot in a `reconnecting state`. However, while the Jackal was trying to reconnect, the Husky robot was turned on by a party not associated with the SMADS system. (SMADS shares robots across groups). The Husky connected to the App Server and began logging status updates. Therefore, to the App Server, the Husky was a viable, available delivery robot. So, when the Jackal failed to connect after 1 min, the App Server changed the Jackal's status to `offline` and redirected the trip to the Husky. The managers reported that the Jackal trip disappeared from the Manager App and was replaced by a trip with the Husky. This reported issue can be explained with the events detailed above, as it is expected behavior that when a robot becomes offline, the robot's order is redirected to the next available robot.

In order to avoid this issue, changes to the App Server's handling of offline robots were made. Instead of redirecting the offline robot's trip to the next available robot, the App Server now assumes that the robot managers will ensure that the robot reaches its destination, whether

it be autonomously or manually driven. This new assumption overrides the original design assumption that this system was operating without human oversight, and therefore, an offline robot had to be assumed missing. Subsequent tests showed that this change resolved the reported issue.

CHAPTER 9

Installation Guide for Future Researchers

This section describes how to install the App Server and the two mobile applications on a development machine for future researchers. The following steps assume development is happening on macOS.

9.1 iOS Applications

The following steps will guide a developer through the setup process for the iOS applications.

(1) Download and install Xcode [1];

(2) Open Xcode and follow the steps to install the additional components. This step will install `Git` which is used to manage the versions of the code;

(3) Clone the project's GitHub repository:
`git clone git@gitlab.com:ashajain/smds_app.git`.
You can also fork the repository if you prefer;

(4) Open the directory where the project was cloned, and double click on `SMADS.xcworkspace`;

(5) Start developing. If Xcode complains about issues signing the build, change the setting to "Personal Team".

Have fun!

[1] https://developer.apple.com/xcode/

9.2 Application Server

The steps below describe the process to start developing the server on a local computer. This project was built using Maven (see section 3.1).

Setting up the database

Let's start by setting up the database.

(1) Install Homebrew [2];
(2) Install MySQL: `brew install mysql`;
(3) Run MySQL to make sure the installation was successful: `mysql -uroot`
(4) Run a script to setup the initial database. Open
`<project directory>/Scripts/DB/init.sql`.
This will create the SMADS user used by the server to connect to the database, as well as the empty database.

At this point, the database should be setup properly.

Setting up the code base in IntelliJ

Let's setup the IDE and the code. The following steps assume the developer is using IntelliJ, a very famous IDE available to users for free using their UT account.

(1) Install the Java Development Kit (JDK):
`brew install openjdk`;
(2) Install Maven:
`brew install maven`;
(3) Install IntelliJ Ultimate[3]

[2]https://docs.brew.sh/Installation
[3]https://www.jetbrains.com/idea/download/#section=mac

(4) Import the project into IntelliJ. IntelliJ should recognize the project was created using Maven and will automatically import the dependencies. If it doesn't, open the Maven tab (View > Tool Windows > Maven) and click the refresh button to force IntelliJ to import the project.

(5) At the top of the screen, click the green triangle to build and start the server. Upon startup, the server will check the state of the database, create the tables if necessary, and insert the initial content.

Tools like Sequel Ace[4] can help visualize the content of the database during development and debugging.

The Maven tab can also help select the proper profile to use when running the server. Corresponding properties are declared in the `*.properties` files.

[4]https://github.com/Sequel-Ace

CHAPTER 10

Future Work

The application development aspect of the SMADS project is a proof of concept. Some of the future work is detailed below.

Future work to the Texas Botler Manager App includes adding a push notification alert when a robot is tampered with. Indeed, in a future iteration of the hardware installed on the delivery robots, the payload would be secured in a container locked with a pin which could be controlled by a computer and shared with the user, so only they can access their order. This would allow the development of a tampering warning system for managers to be notified if anyone wrongfully manipulated the robots or their payload during a delivery.

On the App Server, future work should optimize the trip scheduler. The trip scheduler is used to determine the ETA for each service location and determine which robot will service a given trip. At this time, the scheduler can only assign one trip per robot. One possible improvement would be to allow multi-stop trips, while making sure the robot has enough energy to complete the trip. Optimizing the trip scheduler would also help us improve the ETA calculation. While the current implementation produces relatively accurate results, during the project's testing phase, some of the estimates were 400% off, resulting in poor user experience.

Finally, future research could improve the robot connectivity. Both robots used during the project's testing phase relied on campus WiFi to communicate with the App Server. However, the WiFi connection was not reliable and in several locations, the robots were unable to connect to the Internet. While that did not prevent the robot from operating and continuing their trip, it resulted in the user not receiving the robot's location updates. This issue can be solved by upgrading the robots' hardware to use LTE instead of WiFi to stay connected to the

rest of the system. Furthermore, the Texas Botler app can implement a predictive display that smooths the animation of the robot moving along the delivery route and continues the robot's progression even when the robot has disconnected.

CHAPTER 11

Conclusion

The SMADS system utilizes a variety of subsystems to connect customers to autonomous delivery robots. The customer-facing iOS app, Texas Botler, works to interact with the customer, forwarding customer orders to the App Server and presenting order updates to the customer. The App Server manages these customer orders, schedules deliveries to a specific robot and facilitates communication between the robot and the customer. The Manager App allows robot managers to understand the current state of the SMADS system, fulfill customer orders and send robots back to the robot depot. These three systems work in coordination with the other components of the SMADS systems, namely the robot autonomy stack and the robot servers, to make autonomous robot deliveries possible.

From the presented field results, the SMADS system has shown to implement a successful robot-agnostic app and server management system. This feature enables future researchers to quickly connect their unique robot platforms to a customer-facing iOS app and interact with real users. Future work could build on this app and server management system to study the human-robot interaction during autonomous robot deliveries. More specifically, researchers could study how the customer and other bystanders behave when an autonomous robot stops at its destination and waits for the customer to retrieve the package. Will the bystanders try to engage with the robot? How often do the customers realize that the delivery robot is waiting on them? These questions could motivate future robot design to have displays with the name of the customer, or have speaking capabilities to announce that the order has arrived.

For interested researchers, refer to chapter 9 to review how to install and set up the SMADS app-server management system.

Bibliography

[1] A. Rutter, D. H. Bierling, D. Lee, C. A. Morgan, J. E. Warner *et al.*, "How will e-commerce growth impact our transportation network? final report." Texas A&M Transportation Institute, Tech. Rep., 2017.

[2] M. Figliozzi and D. Jennings, "Autonomous delivery robots and their potential impacts on urban freight energy consumption and emissions," *Transportation Research Procedia*, vol. 46, pp. 21 – 28, 2020, the 11th International Conference on City Logistics, Dubrovnik, Croatia, 12th - 14th June 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352146520303598

[3] G.-Z. Yang, B. Nelson, R. Murphy, H. Choset, H. Christensen, S. Collins, P. Dario, K. Goldberg, K. Ikuta, N. Jacobstein, D. Kragic, R. Taylor, and M. McNutt, "Combating covid-19—the role of robotics in managing public health and infectious diseases," *Science Robotics*, vol. 5, p. eabb5589, 03 2020.

[4] J. Vincent, "Fedex unveils autonomous delivery robot," Feb 2019. [Online]. Available: https://www.theverge.com/2019/2/27/18242834/delivery-robot-fedex-sameday-bot-autonomous-trials

[5] B. Marr, "Demand for these autonomous delivery robots is skyrocketing during this pandemic," May 2020. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2020/05/29/demand-for-these-autonomous-delivery-robots-is-skyrocketing-during-this-pandemic/?sh=3150542c7f3c

[6] J. Biswas and M. Veloso, "Episodic non-markov localization: Reasoning about short-term and long-term features," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 3969–3974.

[7] M. Veloso, J. Biswas, B. Coltin, and S. Rosenthal, "Cobots: Robust symbiotic autonomous mobile service robots," in *Proceedings of the 24th International Conference on*

Artificial Intelligence, ser. IJCAI'15. AAAI Press, 2015, p. 4423–4429.

[8] A. Lappala, "The most in-demand programming technologies used at top us startups," Jul 2019. [Online]. Available: https://www.codingdojo.com/blog/unicorn-languages-report

[9] Flyaps, "See top programming languages big companies prefer," Aug 2020. [Online]. Available: https://flyaps.com/blog/top-10-coding-languages-used-by-global-companies/

[10] JRebel, "2020 java technology report | rebel," Jan 2020. [Online]. Available: https://www.jrebel.com/blog/2020-java-technology-report

[11] "Welcome to the apache software foundation!" 2020. [Online]. Available: https://www.apache.org/

[12] M. Dabbs, "How web apps work - web application architecture simplified," Jul 2019. [Online]. Available: https://reinvently.com/blog/fundamentals-web-application-architecture/

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reuseable object - oriented software*. Addison Wesley, 2000.

[14] P. e. Todd Fredrich, "Learn rest: A restful tutorial," 2020. [Online]. Available: https://www.restapitutorial.com/

[15] A. vyas, "Mvc pattern," Oct 2018. [Online]. Available: https://medium.com/@anshul.vyas380/mvc-pattern-3b5366e60ce4

[16] MuleSoft, "Microservices vs monolithic architecture," 2020. [Online]. Available: https://www.mulesoft.com/resources/api/microservices-vs-monolithic

[17] M. Ryan, "Microservices - the robust architecture for applications." [Online]. Available: https://www.royalcyber.com/blog/cloud/microservices-the-robust-architecture-for-applications/

[18] T. Foltyn, "Majority of the world's top million websites now use https," Sep 2018. [Online]. Available: https://www.welivesecurity.com/2018/09/03/majority-worlds-top-websites-https/

[19] Auth0, "Json web tokens introduction." [Online]. Available: https://jwt.io/introduction/

[20] "Jwt security best practices." [Online]. Available: https://curity.io/resources/architect/api-security/jwt-best-practices/

[21] A. Inc., "Testflight," 2020. [Online]. Available: https://developer.apple.com/testflight/

[22] C. Hale, "How to safely store a password," Jan 2010. [Online]. Available: https://codahale.com/how-to-safely-store-a-password/

[23] R. Heaton, "How does https actually work?" Mar 2014. [Online]. Available: https://robertheaton.com/2014/03/27/how-does-https-actually-work/

1 Appendix A

The following Appendix contains the documentation of all of the Application Server's REST API endpoints, including the models for each objects they require.

APIs for the SMADS Application Server

Overview

This page lists all the REST APIs used to develop the Application Server

URI scheme

Host : localhost:8080
BasePath : /

Tags

- admin-authentication-controller : Admin Authentication Controller
- authentication-controller : Authentication Controller
- feedback-controller : Feedback Controller
- manager-controller : Manager Controller
- notification-controller : Notification Controller
- request-controller : Request Controller
- service-location-controller : Service Location Controller
- spot-controller : Spot Controller
- store-controller : Store Controller
- test-communication-controller : Test Communication Controller
- user-controller : User Controller

Paths

createManager

```
POST /admin/auth/signupManager
```

Parameters

Type	Name	Description	Schema
Body	**authenticationRequest** *required*	authenticationRequest	AuthenticationRequest

Responses

HTTP Code	Description	Schema
200	OK	AuthenticationResponse
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- admin-authentication-controller

getAllUsers

```
GET /auth/
```

Responses

HTTP Code	Description	Schema
200	OK	AllActiveUsersResponse

HTTP Code	Description	Schema
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- authentication-controller

login

```
POST /auth/login
```

Parameters

Type	Name	Description	Schema
Body	**authenticationRequest** *required*	authenticationRequest	AuthenticationRequest

Responses

HTTP Code	Description	Schema
200	OK	AuthenticationResponse
201	Created	No Content
401	Unauthorized	No Content

HTTP Code	Description	Schema
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- authentication-controller

registerUser

```
POST /auth/registerGoogleUser
```

Parameters

Type	Name	Description	Schema
Body	**request** *required*	request	GoogleAuthenticationRequest

Responses

HTTP Code	Description	Schema
200	OK	AuthenticationResponse
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- authentication-controller

createCustomer

```
POST /auth/signup
```

Parameters

Type	Name	Description	Schema
Body	**authenticationRequest** *required*	authenticationRequest	AuthenticationRequest

Responses

HTTP Code	Description	Schema
200	OK	AuthenticationResponse
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- */*

Tags

- authentication-controller

isTokenValidForCustomerAndManager

```
POST /auth/validateToken
```

Responses

HTTP Code	Description	Schema
200	OK	ValidTokenResponse
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- */*

Tags

- authentication-controller

deleteUser

```
DELETE /auth/{username}
```

Parameters

Type	Name	Description	Schema
Path	**username** *required*	username	string

Responses

HTTP Code	Description	Schema
200	OK	boolean
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- authentication-controller

createFeedback

```
POST /feedback
```

Parameters

Type	Name	Description	Schema
Body	**feedback** *required*	feedback	Feedback

Responses

HTTP Code	Description	Schema
200	OK	Feedback
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- feedback-controller

getAllIssues

```
GET /feedback/issues
```

Responses

HTTP Code	Description	Schema
200	OK	AllIssuesResponse
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- */*

Tags

- feedback-controller

createNewManager

```
POST /managers/
```

Parameters

Type	Name	Description	Schema
Body	**newManagerRequest** *required*	newManagerRequest	NewManagerRequest

Responses

HTTP Code	Description	Schema
200	OK	Manager
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- */*

Tags

- manager-controller

getAllManagers

```
GET /managers/
```

Responses

HTTP Code	Description	Schema
200	OK	AllManagersResponse
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- manager-controller

deleteManager

```
DELETE /managers/
```

Parameters

Type	Name	Description	Schema
Body	**request** *required*	request	DeleteManagerRequest

Responses

HTTP Code	Description	Schema
200	OK	DeleteManagerResponse
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- manager-controller

sendTestNotification

```
POST /notifications/send
```

Responses

HTTP Code	Description	Schema
200	OK	No Content
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- notification-controller

addTokenForUser

```
POST /notifications/tokens
```

Parameters

Type	Name	Description	Schema
Body	**addTokenRequest** *required*	addTokenRequest	AddTokenRequest

Responses

HTTP Code	Description	Schema
200	OK	SimpleSuccessResponse
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- */*

Tags

- notification-controller

getTokensForUser

```
GET /notifications/tokens
```

Responses

HTTP Code	Description	Schema
200	OK	< string > array
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- application/json

Produces

- */*

Tags

- notification-controller

createTrip

```
POST /requests/
```

Parameters

Type	Name	Description	Schema
Body	**tripRequest** *required*	tripRequest	TripRequest

Responses

HTTP Code	Description	Schema
200	OK	NewRequestResponse
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- request-controller

getAllTripsToComplete

```
GET /requests/notcomplete
```

Responses

HTTP Code	Description	Schema
200	OK	TripsToCompleteResponse

HTTP Code	Description	Schema
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- request-controller

completeTrip

```
PUT /requests/{tripID}/complete
```

Parameters

Type	Name	Description	Schema
Path	**tripID** *required*	tripID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	boolean
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content

HTTP Code	Description	Schema
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- request-controller

doesTripHaveRobot

```
GET /requests/{tripID}/hasRobot
```

Parameters

Type	Name	Description	Schema
Path	**tripID** *required*	tripID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	Trip
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- */*

Tags

- request-controller

getSpotAssignedToTrip

```
GET /requests/{tripId}
```

Parameters

Type	Name	Description	Schema
Path	**tripId** *required*	tripId	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	Spot
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- application/json

Produces

- */*

Tags

- request-controller

deleteTrip

```
DELETE /requests/{tripId}
```

Parameters

Type	Name	Description	Schema
Path	**tripId** *required*	tripId	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	boolean
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- request-controller

notifyUserTripHasArrived

```
POST /requests/{tripId}/arrived
```

Parameters

Type	Name	Description	Schema
Path	**tripId** *required*	tripId	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	No Content
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- request-controller

sendRobotOnTrip

```
POST /requests/{tripId}/send
```

Parameters

Type	Name	Description	Schema
Path	**tripId** *required*	tripId	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	No Content
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- request-controller

getTripStatus

```
GET /requests/{tripId}/status
```

Parameters

Type	Name	Description	Schema
Path	**tripId** *required*	tripId	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	Trip

HTTP Code	Description	Schema
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- request-controller

saveServiceLocation

```
POST /serviceLocations/
```

Parameters

Type	Name	Description	Schema
Body	**serviceLocation** *required*	serviceLocation	ServiceLocation

Responses

HTTP Code	Description	Schema
200	OK	ServiceLocation
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content

HTTP Code	Description	Schema
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- service-location-controller

getAllServiceLocations

```
GET /serviceLocations/
```

Parameters

Type	Name	Description	Schema
Query	**shouldCalculateETA** *optional*	shouldCalculateETA	boolean
Query	**type** *optional*	type	enum (library, officebuilding, dorm, restaurant, other)

Responses

HTTP Code	Description	Schema
200	OK	AvailableServiceLocationsResponse
401	Unauthorized	No Content
403	Forbidden	No Content

HTTP Code	Description	Schema
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- service-location-controller

reactivateServiceLocation

```
PUT /serviceLocations/{databaseID}
```

Parameters

Type	Name	Description	Schema
Path	**databaseID** *required*	databaseID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	ServiceLocation
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- service-location-controller

deactivateServiceLocation

```
DELETE /serviceLocations/{databaseID}
```

Parameters

Type	Name	Description	Schema
Path	**databaseID** *required*	databaseID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	ServiceLocation
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- service-location-controller

getServiceLocationWithLocationName

```
GET /serviceLocations/{locationName}
```

Parameters

Type	Name	Description	Schema
Path	**locationName** *required*	locationName	string

Responses

HTTP Code	Description	Schema
200	OK	ServiceLocation
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- service-location-controller

createSpot

```
POST /spots/
```

Parameters

Type	Name	Description	Schema
Body	**newSpotRequest** *required*	newSpotRequest	SpotCreationRequest

Responses

HTTP Code	Description	Schema
200	OK	Spot
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

getListOfSpots

```
GET /spots/
```

Responses

HTTP Code	Description	Schema
200	OK	AllSpotResponse
401	Unauthorized	No Content

HTTP Code	Description	Schema
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

getSpotById

```
GET /spots/{manufacturerID}
```

Parameters

Type	Name	Description	Schema
Path	**manufacturerID** *required*	manufacturerID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	Spot
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

deleteSpot

```
DELETE /spots/{manufacturerID}
```

Parameters

Type	Name	Description	Schema
Path	**manufacturerID** *required*	manufacturerID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	boolean
204	No Content	No Content
401	Unauthorized	No Content
403	Forbidden	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

getActiveTripForSpot

```
GET /spots/{manufacturerID}/activeTrip
```

Parameters

Type	Name	Description	Schema
Path	**manufacturerID** *required*	manufacturerID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	Trip
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

sendSpotHome

```
POST /spots/{manufacturerID}/returnHome
```

Parameters

Type	Name	Description	Schema
Path	**manufacturerID** *required*	manufacturerID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	boolean
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

getUpdatedSpotCondition

```
GET /spots/{manufacturerID}/statusUpdate
```

Parameters

Type	Name	Description	Schema
Path	**manufacturerID** *required*	manufacturerID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	UpdatedSpotConditionResponse
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

updateSpotStatus

```
PUT /spots/{manufacturerID}/statusUpdate
```

Parameters

Type	Name	Description	Schema
Path	**manufacturerID** *required*	manufacturerID	integer (int32)

Type	Name	Description	Schema
Body	**updatedSpotCondition** *required*	updatedSpotCondition	NewSpotConditionRequest

Responses

HTTP Code	Description	Schema
200	OK	UpdatedSpotConditionResponse
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

getTripsForSpot

```
GET /spots/{manufacturerID}/trips
```

Parameters

Type	Name	Description	Schema
Path	**manufacturerID** *required*	manufacturerID	integer (int32)

Responses

HTTP Code	Description	Schema
200	OK	AllTripsResponse
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

updateSpotStatusVariable

```
PUT /spots/{manufacturerID}/updateSpotStatus
```

Parameters

Type	Name	Description	Schema
Path	**manufacturerID** *required*	manufacturerID	integer (int32)
Body	**request** *required*	request	UpdateSpotStatusRequest

Responses

HTTP Code	Description	Schema
200	OK	boolean

HTTP Code	Description	Schema
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- spot-controller

updateStoreDescription

```
POST /stores/description
```

Parameters

Type	Name	Description	Schema
Body	**request** *required*	request	StoreDescriptionRequest

Responses

HTTP Code	Description	Schema
200	OK	StoreStatusResponse
201	Created	No Content
401	Unauthorized	No Content

HTTP Code	Description	Schema
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- store-controller

updateStoreStatus

```
POST /stores/status
```

Parameters

Type	Name	Description	Schema
Body	**request** *required*	request	StoreStatusRequest

Responses

HTTP Code	Description	Schema
200	OK	StoreStatusResponse
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- application/json

Produces

- */*

Tags

- store-controller

getStoreStatus

```
GET /stores/status
```

Responses

HTTP Code	Description	Schema
200	OK	StoreStatusResponse
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- application/json

Produces

- */*

Tags

- store-controller

testSSLCommunication

```
GET /testComms/ssl
```

Responses

HTTP Code	Description	Schema
200	OK	boolean
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- test-communication-controller

getActiveTripForCustomer

```
GET /users/activeTrip
```

Responses

HTTP Code	Description	Schema
200	OK	Trip
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- */*

Tags

- user-controller

getAllTripsForCustomer

```
GET /users/trips
```

Responses

HTTP Code	Description	Schema
200	OK	AllTripsResponse
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- application/json

Produces

- */*

Tags

- user-controller

updateUserInfo

```
PUT /users/updateUserInfo
```

Parameters

Type	Name	Description	Schema
Body	**newUpdateUserInfo** *required*	newUpdateUserInfo	User

Responses

HTTP Code	Description	Schema
200	OK	User
201	Created	No Content
401	Unauthorized	No Content
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- user-controller

getUserInfo

```
GET /users/userInfo
```

Responses

HTTP Code	Description	Schema
200	OK	User
401	Unauthorized	No Content

HTTP Code	Description	Schema
403	Forbidden	No Content
404	Not Found	No Content

Consumes

- `application/json`

Produces

- `*/*`

Tags

- user-controller

Definitions

AddTokenRequest

Name	Schema
manager *optional*	boolean
token *optional*	string

AllActiveUsersResponse

Name	Schema
userList *optional*	< User > array

AllIssuesResponse

Name	Schema
issues *optional*	< Issue > array

AllManagersResponse

Name	Schema
allManagers *optional*	< Manager > array

AllSpotResponse

Name	Schema
spots *optional*	< Spot > array

AllTripsResponse

Name	Schema
allTrips *optional*	< Trip > array

AuthenticationRequest

Name	Schema
name *optional*	string
password *optional*	string
username *optional*	string

AuthenticationResponse

Name	Schema
customerTrip *optional*	Trip
isManager *optional*	boolean

Name	Schema
manager *optional*	boolean
token *optional*	string

AvailableServiceLocationsResponse

Name	Schema
serviceLocationList *optional*	< ServiceLocation > array

Chronology

Name	Schema
calendarType *optional*	string
id *optional*	string

CollectionÂ«IssueÂ»

Type : object

DeleteManagerRequest

Name	Schema
emailAddress *optional*	string

DeleteManagerResponse

Name	Schema
successfullyDeleted *optional*	boolean

Duration

Name	Schema
nano *optional*	integer (int32)
negative *optional*	boolean
seconds *optional*	integer (int64)
units *optional*	< TemporalUnit > array
zero *optional*	boolean

Feedback

Name	Schema
comment *optional*	string
issues *optional*	CollectionÂ«IssueÂ»
rating *optional*	integer (int32)
tripID *optional*	integer (int32)

GoogleAuthenticationRequest

Name	Schema
idToken *optional*	string

Instant

Name	Schema
epochSecond *optional*	integer (int64)
nano *optional*	integer (int32)

Issue

Name	Schema
id *optional*	integer (int64)
issue *optional*	string
updated *optional*	ZonedDateTime

LocalDateTime

Name	Schema
chronology *optional*	Chronology
dayOfMonth *optional*	integer (int32)
dayOfWeek *optional*	enum (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY)
dayOfYear *optional*	integer (int32)
hour *optional*	integer (int32)
minute *optional*	integer (int32)

Name	Schema
month *optional*	enum (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER)
monthValue *optional*	integer (int32)
nano *optional*	integer (int32)
second *optional*	integer (int32)
year *optional*	integer (int32)

LocalTime

Name	Schema
hour *optional*	integer (int32)
minute *optional*	integer (int32)
nano *optional*	integer (int32)
second *optional*	integer (int32)

Manager

Name	Schema
emailAddress *optional*	string

NewManagerRequest

Name	Schema
emailAddress *optional*	string

NewRequestResponse

Name	Schema
goToActiveTripDirectly *optional*	boolean
trip *optional*	Trip
userHasTrip *optional*	boolean

NewSpotConditionRequest

Name	Schema
chargeLevel *optional*	number (double)
heading *optional*	number (double)
latitude *optional*	number (double)
longitude *optional*	number (double)
spotStatus *optional*	enum (enroute, pickup, dropoff, charging, available, outofservice, returninghome, atHome, reconnectingToInternet, assignedTrip)
timestamp *optional*	StatusTimestamp

NotificationSent

Name	Schema
id *optional*	integer (int32)
type *optional*	enum (pickup, robotenroute, robotassigned)

ServiceLocation

Name	Schema
acronym *optional*	string
active *optional*	boolean
eta *optional*	integer (int32)
home *optional*	boolean
id *optional*	integer (int32)
latitude *optional*	number (double)
locationName *optional*	string
locationType *optional*	enum (library, officebuilding, dorm, restaurant, other)
longitude *optional*	number (double)
numAvailableChargers *optional*	integer (int32)

SimpleSuccessResponse

Name	Schema
success *optional*	boolean

Spot

Name	Schema
active *optional*	boolean
chargeLevel *optional*	number (double)
currentLatitude *optional*	number (double)
currentLongitude *optional*	number (double)
heading *optional*	number (double)
id *optional*	integer (int32)
ipAddress *optional*	string
manufacturerID *optional*	integer (int32)
name *optional*	string
status *optional*	enum (enroute, pickup, dropoff, charging, available, outofservice, returninghome, atHome, reconnectingToInternet, assignedTrip)
updatedAt *optional*	ZonedDateTime

SpotCreationRequest

Name	Schema
ipAddress *optional*	string
manufacturerID *optional*	integer (int32)
name *optional*	string
password *optional*	string
spotId *optional*	integer (int32)

StatusTimestamp

Name	Schema
milliseconds *optional*	number (double)
seconds *optional*	number (double)

Store

Name	Schema
hoursDescription *optional*	string
id *optional*	integer (int32)
name *optional*	string
open *optional*	boolean

StoreDescriptionRequest

Name	Schema
description *optional*	string

StoreStatusRequest

Name	Schema
open *optional*	boolean

StoreStatusResponse

Name	Schema
store *optional*	Store

TemporalUnit

Name	Schema
dateBased *optional*	boolean
durationEstimated *optional*	boolean
timeBased *optional*	boolean

Trip

Name	Schema
active *optional*	boolean
assignedSpot *optional*	Spot

Name	Schema
dropoffLocation *optional*	ServiceLocation
endTime *optional*	ZonedDateTime
eta *optional*	integer (int32)
id *optional*	integer (int32)
payloadContent *optional*	string
pickupLocation *optional*	ServiceLocation
sentNotifications *optional*	< NotificationSent > array
spotManufacturerID *optional*	integer (int32)
startTime *optional*	ZonedDateTime
tripStatus *optional*	enum (requested, enroute, dropoff, returningHome, complete, cancelled, processing)
userID *optional*	integer (int32)
username *optional*	string
waypoints *optional*	< Waypoint > array

TripRequest

Name	Schema
dropoffLocID *optional*	integer (int32)
eta *optional*	integer (int32)
payloadContent *optional*	string
pickupLocID *optional*	integer (int32)

TripsToCompleteResponse

Name	Schema
activeTrips *optional*	< Trip > array
returningHomeTrips *optional*	< Trip > array
tripsToBeCompleted *optional*	< Trip > array

UpdateSpotStatusRequest

Name	Schema
status *optional*	enum (enroute, pickup, dropoff, charging, available, outofservice, returninghome, atHome, reconnectingToInternet, assignedTrip)

UpdatedSpotConditionResponse

Name	Schema
chargeLevel *optional*	number (double)
heading *optional*	number (double)

Name	Schema
manufacturerId *optional*	integer (int32)
spotStatus *optional*	enum (enroute, pickup, dropoff, charging, available, outofservice, returninghome, atHome, reconnectingToInternet, assignedTrip)
timestamp *optional*	StatusTimestamp
updatedSpotLatitude *optional*	number (double)
updatedSpotLongitude *optional*	number (double)

User

Name	Schema
active *optional*	boolean
firstName *optional*	string
id *optional*	integer (int32)
lastName *optional*	string
manager *optional*	boolean
username *optional*	string

ValidTokenResponse

Name	Schema
activeTrip *optional*	Trip
customer *optional*	boolean

Waypoint

Name	Schema
id *optional*	integer (int32)
latitude *optional*	number (double)
longitude *optional*	number (double)

ZoneId

Name	Schema
id *optional*	string
rules *optional*	ZoneRules

ZoneOffset

Name	Schema
id *optional*	string
rules *optional*	ZoneRules
totalSeconds *optional*	integer (int32)

ZoneOffsetTransition

Name	Schema
dateTimeAfter *optional*	LocalDateTime
dateTimeBefore *optional*	LocalDateTime
duration *optional*	Duration
gap *optional*	boolean
instant *optional*	Instant
overlap *optional*	boolean

ZoneOffsetTransitionRule

Name	Schema
dayOfMonthIndicator *optional*	integer (int32)
dayOfWeek *optional*	enum (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY)
localTime *optional*	LocalTime
midnightEndOfDay *optional*	boolean
month *optional*	enum (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER)
timeDefinition *optional*	enum (UTC, WALL, STANDARD)

ZoneRules

Name	Schema
fixedOffset *optional*	boolean
transitionRules *optional*	< ZoneOffsetTransitionRule > array
transitions *optional*	< ZoneOffsetTransition > array

ZonedDateTime

Name	Schema
chronology *optional*	Chronology
dayOfMonth *optional*	integer (int32)
dayOfWeek *optional*	enum (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY)
dayOfYear *optional*	integer (int32)
hour *optional*	integer (int32)
minute *optional*	integer (int32)
month *optional*	enum (JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER)
monthValue *optional*	integer (int32)
nano *optional*	integer (int32)

Name	Schema
offset *optional*	ZoneOffset
second *optional*	integer (int32)
year *optional*	integer (int32)
zone *optional*	ZoneId